Automated Control of Multiple Virtualized Resources

Pradeep Padala,
Kai-Yuan Hou, Kang G. Shin
The University of Michigan
{ppadala,karenhou,kgshin}@umich.edu

Xiaoyun Zhu *
VMware, Inc.
xzhu@vmware.com

Mustafa Uysal, Zhikui Wang,
Sharad Singhal, Arif Merchant
HP Labs
firstname.lastname@hp.com

Abstract

Virtualized data centers enable sharing of resources among hosted applications. However, it is difficult to satisfy service-level objectives (SLOs) of applications on shared infrastructure, as application workloads and resource consumption patterns change over time. In this paper, we present *AutoControl*, a resource control system that automatically adapts to dynamic workload changes to achieve application SLOs. *AutoControl* is a combination of an online model estimator and a novel multi-input, multi-output (MIMO) resource controller. The model estimator captures the complex relationship between application performance and resource allocations, while the MIMO controller allocates the right amount of multiple virtualized resources to achieve application SLOs. Our experimental evaluation with RUBiS and TPC-W benchmarks along with production-trace-driven workloads indicates that *AutoControl* can detect and mitigate CPU and disk I/O bottlenecks that occur over time and across multiple nodes by allocating each resource accordingly. We also show that *AutoControl* can be used to provide service differentiation according to the application priorities during resource contention.

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

Keywords Data center, server consolidation, virtualization, control theory, automated control, application QoS, resource management

* Xiaoyun Zhu worked on this project while employed at HP labs.

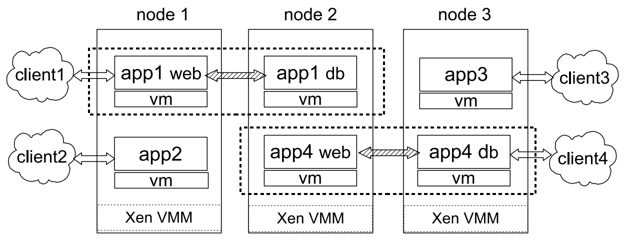


Figure 1. An example for *shared virtualized infrastructure*: Each physical node hosts multiple application tiers running on VMs; Multi-tier applications can span multiple nodes.

General Terms DESIGN, EXPERIMENTATION, MANAGEMENT, PERFORMANCE

1. Introduction

Virtualization is causing a disruptive change in enterprise data centers and giving rise to a new paradigm: *shared virtualized infrastructure*. Figure 1 shows a three-node subset of a virtualized infrastructure shared by multiple applications, where each tier of an application is hosted in a virtual machine (VM), and a multi-tier application (such as app1 or app4) may span multiple nodes. Unlike the traditional hosting model where applications run on dedicated nodes, resulting in low resource utilization, this model allows applications to be consolidated onto fewer nodes, reducing capital expenditure on infrastructure as well as operating costs on power, cooling, maintenance, and support. It also leads to much higher resource utilization on the shared nodes.

Data center administrators need to meet the service-level objectives (SLOs) for the hosted applications despite changing resource requirements and unpredictable interactions between the applications. They face several challenges:

- *Complex SLOs*: It is non-trivial to convert individual application SLOs to resource allocations on the virtualized nodes. For example, it is difficult to determine the CPU and disk allocations needed to process a specified number of financial transactions per unit of time.
- *Time-varying resource requirements*: The intensity and the mix of typical enterprise application workloads change over the lifetime of an application. As a result, the de-

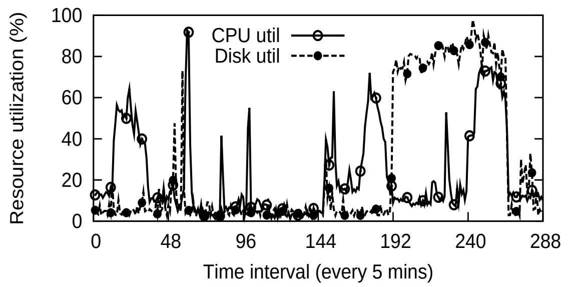


Figure 2. Average CPU utilization and peak disk utilization measured every 5 minutes in a production SAP application server for a 24-hour period.

mands for individual resources also change over time. For example, Figure 2 shows the CPU and disk utilization of a production SAP application for a 24-hour period. The utilization for both resources varied over time considerably, and the peaks of the two resource types occurred at different times of the day. This implies that static resource allocation can meet application SLOs only when the resources are allocated for peak demands, thus wasting resources.

- *Distributed resource allocations*: Multi-tier applications spanning multiple nodes require resource allocations across all tiers to be at appropriate levels to meet end-to-end application SLOs.
- *Resource dependencies*: Application-level performance often depends on the application's ability to simultaneously access multiple types of system-level resources.

Researchers have studied capacity planning for such an environment by using historical resource utilization traces to predict application resource requirements in the future and to place compatible sets of applications onto shared nodes [Rolia 2005]. This approach aims to ensure that each node has enough capacity to meet the aggregate demand of all the applications, while minimizing the number of active nodes. However, past demands are not always accurate predictors of future demands, especially for Web-based, interactive applications. Furthermore, in a shared virtualized infrastructure, the performance of a given application depends on other applications sharing resources, making it difficult to predict its behavior using pre-consolidation traces. Other researchers have considered use of live VM migration to alleviate overload conditions that occur at runtime [Wood 2007]. However, the CPU and network overheads of VM migration may further degrade application performance on the already-congested node, and hence, VM migration is mainly effective for sustained, rather than transient, overload.

In this paper, we propose *AutoControl*, a feedback-based resource allocation system that manages dynamic resource sharing within the virtualized nodes. *AutoControl* complements the capacity planning and workload migration schemes others have proposed to achieve application-level SLOs on shared virtualized infrastructure.

Our main contributions are twofold. First, we design an *online model estimator* to dynamically determine the relationship between application-level performance and the allocations of individual resources. Our adaptive modeling approach captures the complex behavior of enterprise applications including time-varying resource demands, resource demands from distributed application tiers, and shifting demands across multiple resource types. Second, we design a two-layer, *multi-input, multi-output (MIMO) controller* to automatically allocate multiple types of virtualized resources to individual tiers of enterprise applications to achieve their SLOs. The first layer consists of a set of *application controllers* that automatically determine the amount of resources necessary to achieve individual application SLOs, using the estimated models and a feedback-based approach. The second layer is comprised of a set of *node controllers* that detect resource bottlenecks on the shared nodes and properly allocate multiple types of resources to individual application tiers. Under overload, the node controllers provide service differentiation according to the priorities of individual applications.

We have built two testbeds using Xen [Barham 2003] to evaluate *AutoControl* in various scenarios. Our experimental results show that, (i) *AutoControl* can detect and adapt to bottlenecks in both CPU and disk across multiple nodes; (ii) the MIMO controller can handle multiple multi-tier applications running RUBiS and TPC-W benchmarks along with workloads driven by production traces, and provide better performance than work-conserving and static allocation methods; and (iii) priorities can be enforced among different applications during resource contention.

The remainder of the paper is organized as follows. Section 2 provides an overview of *AutoControl*. This is followed by a detailed description of the design of the model estimator and the MIMO controller in Section 3. Section 4 discusses experimental methodology and testbed setup. We present experimental evaluation results in Section 5, followed by discussion and future work in Section 6. Section 7 discusses related work, and finally, conclusions are drawn in Section 8.

2. Overview, Assumptions and Goals

In *AutoControl*, operators specify the application SLO as a tuple $(priority, metric, target)$, where *priority* represents the priority of the application, *metric* specifies the performance metric in the SLO (e.g., transaction throughput, response time), and *target* indicates the desired value for the performance metric. Currently, our implementation supports only a single metric specification at a time, but the architecture can be generalized to support different metrics for different applications. *AutoControl* can manage any resource that affects the performance metric of interest and that can be allocated among the applications. In this paper, we use CPU and disk

I/O as the resources, and application throughput or average response time as the performance metric.

A fundamental assumption behind *AutoControl* is that the initial placement of applications onto nodes has been handled by a separate capacity planning service. The same service can also perform admission control for new applications and determine if existing nodes have enough idle capacity to accommodate the projected demands of the applications [Rolia 2005]. We also assume that a workload migration system [Wood 2007] may rebalance workloads among nodes at a time scale of minutes or longer. These problems are complementary to the problem solved by *AutoControl*. Our system deals with runtime management of applications sharing virtualized nodes, and dynamically adjusts resource allocations over short time scales (e.g., seconds) to meet the application SLOs. When any resource on a shared node becomes a bottleneck due to unpredicted spikes in some workloads, or because of complex interactions between co-hosted applications, *AutoControl* provides performance differentiation so that higher-priority applications experience lower performance degradation.

We set the following goals in designing *AutoControl*:

Performance assurance: If all applications can meet their individual performance targets, *AutoControl* should allocate resources to achieve them; otherwise, *AutoControl* should provide service differentiation according to application priorities.

Automation: While performance targets and certain parameters within *AutoControl* may be set manually, all allocation decisions should be made *automatically*.

Adaptation: The controller should adapt to variations in workloads or system conditions.

Scalability: The controller architecture should be distributed so that it can handle many applications and nodes.

Based on these principles, we have designed *AutoControl* with a two-layer, distributed architecture, including a set of *application controllers* (AppControllers) and a set of *node controllers* (NodeControllers). There is one AppController for each hosted application, and one NodeController for each virtualized node. Figure 3 shows the logical controller architecture for the system shown in Figure 1.

For each application, its AppController periodically polls an application performance sensor for the measured performance. We refer to this period as the *control interval*. The AppController compares this measurement with the application performance target. Based on the discrepancy, it determines the resource allocations needed for the next control interval and sends these requests to the NodeControllers for the nodes hosting the individual tiers of the application.

For each node, based on the collective requests from all relevant AppControllers, the corresponding NodeController determines whether it has enough resource of each type to satisfy all demands, and computes the actual resource allocations using the methods described in Section 3. The computed allocation values are fed into the resource schedulers in the virtualization layer for actuation, which allocate the corresponding portions of the node's resources to the VMs. Figure 3 shows the CPU and disk schedulers as examples.

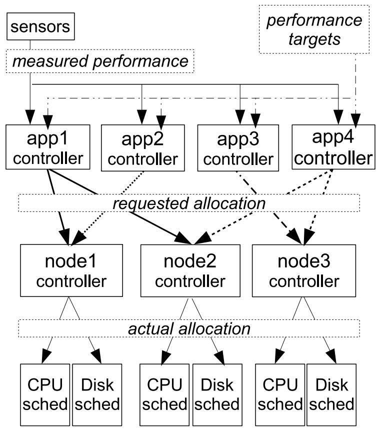


Figure 3. Logical controller architecture: Each application has an application controller that determines the application's resource needs; each node has a node controller that arbitrates the requests from multiple application controllers.

The *AutoControl* architecture allows the placement of AppControllers and NodeControllers in a distributed fashion. For example, each NodeController can be hosted in the physical node it controls, and each AppController in a node where one of its application tiers is located. We do not mandate this placement, however, and the data center operator can choose to host a set of controllers in a node dedicated for control operations. We assume that all nodes in the data center are connected with a high speed network, so that sensing and actuation delays within *AutoControl* are small compared to the control interval.

3. Design

In the following subsections, we will describe the internals of the AppController and the NodeController. Each application has an AppController, whose goal is to compute the amount of resources required for the application to meet its performance target independent of other applications. Each AppController forwards the estimated resource requirements to related NodeControllers. Each node has a NodeController, which arbitrates among multiple applications sharing the node based on its resource availability and individual application priorities.

For easy reference, Table 1 summarizes the mathematical symbols that will be used for key parameters and variables in these controllers.

3.1 Design of AppController

As introduced in Section 2, every application has an AppController associated with it. Every AppController has two modules as illustrated in Figure 4: (1) a *model estimator* that

Table 1. Notation

A	set of all hosted applications
T_a	set of all the tiers in application $a \in A$, e.g., $T_a = \{web, db\}$
R	set of all resource types controlled, e.g., $R = \{cpu, disk\}$
$x(k)$	value of variable x in control interval k
$\bar{u}_{a,r,t}$	requested allocation of resource type r to tier t of application a, $0 \leq \bar{u}_{a,r,t}(k) \leq 1$ ($\bar{u}_{a,r}$ for single-tier applications)
$u_{a,r,t}$	actual allocation of resource type r to tier t of application a, $0 \leq u_{a,r,t}(k) \leq 1$ ($u_{a,r}$ for single-tier applications)
y_a	measured performance of application a
$\bar{y}_a$	performance target for application a
$\hat{y}_a$	normalized performance for application a, where $\hat{y}_a = y_a/\bar{y}_a$
w_a	priority weight for application a
q	stability factor in the AppController

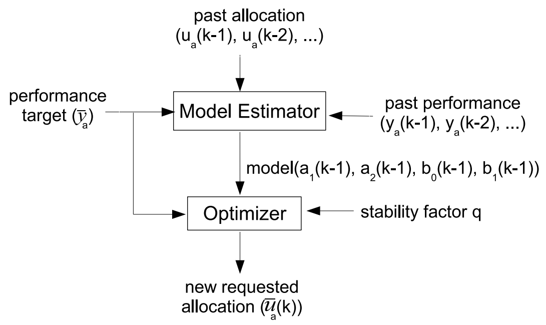

past allocation
(u_a(k-1), u_a(k-2), ...)
performance target ($\bar{y}_a$)
Model Estimator
past performance
(y_a(k-1), y_a(k-2), ...)
model(a_1(k-1), a_2(k-1), b_0(k-1), b_1(k-1))
Optimizer
stability factor q
new requested allocation ($\bar{u}_a$(k))

Figure 4. AppController's internal structure

automatically learns and periodically updates a model for the dynamic relationship between the application's resource allocations and its performance under the current operating conditions, and (2) an *optimizer* that predicts the resource allocations required for the application to meet its performance target based on the estimated model.

To simplify the notation, for each application $a \in A$, we define the resource-allocation vector $\mathbf{u}_a$ (all vectors are in boldface) to represent all the resource allocations for application a that are being managed by the controller. That means, $\mathbf{u}_a$ contains all the elements in the set $\{u_{a,r,t} : r \in R, t \in T_a\}$. For example, for a two-tier application whose performance depends on two critical resources, e.g., $T_a = \{web, db\}$ and $R = \{cpu, disk\}$, $\mathbf{u}_a$ is a 4-dimensional vector. $\mathbf{u}_a(k)$ represents the resource-allocation values for application a during interval k.

3.1.1 Model estimator

In classical control theory, "transfer functions" have been used to model the dynamic relationship between a set of metrics and a set of control knobs for physical systems. However, most computing systems, such as the one considered in this paper, cannot be represented by a single, linear transfer function (or model) because their behavior is often nonlinear and workload-dependent. We assume, however, that the behavior of the system in the neighborhood of an operating point can be approximated *locally* by a linear model. We periodically re-estimate the model based on real-time measurements of the relevant variables and metrics, allowing the model to adapt to different operating points and workloads.

For every control interval, the model estimator re-computes a linear model that approximates the quantitative relationship between the resource allocations to application a ($\mathbf{u}_a$) and its normalized performance ($\hat{y}_a$) around the current operating point. More specifically, the following auto-regressive-moving-average (ARMA) model is used to represent this relationship:

$$\begin{aligned} \hat{y}_a(k) = \quad & a_1(k)\,\hat{y}_a(k-1) + a_2(k)\,\hat{y}_a(k-2) \\ & + \mathbf{b_0}^T(k)\mathbf{u}_a(k) + \mathbf{b_1}^T(k)\mathbf{u}_a(k-1), \end{aligned} \tag{1}$$

where the model parameters $a_1(k)$ and $a_2(k)$ capture the correlation between the application's past and present performance, and $\mathbf{b_0}(k)$ and $\mathbf{b_1}(k)$ are vectors of coefficients capturing the correlation between the current performance and the recent resource allocations. Both $\mathbf{u}_a(k)$ and $\mathbf{u}_a(k-1)$ are column vectors, and $\mathbf{b_0}^T(k)$ and $\mathbf{b_1}^T(k)$ are row vectors. In our experiments, we have found that the second-order ARMA model in Eq. (1) (i.e., one that takes into account the past two control intervals) can predict the application performance with adequate accuracy. This applies to cases where either throughput or response time is used as the performance metric.

We model the normalized performance rather than the absolute performance because the latter can have an arbitrary magnitude. Using the normalized performance $\hat{y}_a$, which has values between 0 and 1, comparable to those of the resource allocations in $\mathbf{u}_a$, improves the numerical stability of the algorithm.

Note that the model represented in Eq. (1) is itself *adaptive*, because the model parameters a_1, a_2, $\mathbf{b_0}$ and $\mathbf{b_1}$ are also functions of control interval k. These parameters are updated at the end of every interval k using the recursive least squares (RLS) method [Astrom 1995], when the measurement for the normalized performance $\hat{y}_a(k)$ for that interval becomes available. The approach assumes that drastic variations in workloads that cause significant changes in the underlying model parameters occur infrequently relative to the control interval, thus allowing the model to converge before the operating point changes significantly. The recursive nature of the RLS algorithm makes the time taken for this computation negligible (average of 10ms in our implementation), as the model is updated recursively instead of being computed from scratch every interval.

3.1.2 Optimizer

The main goal of the optimizer is to determine the resource allocations required ($\bar{\mathbf{u}}_\mathbf{a}$) for the application to meet its performance target. An additional goal is to accomplish this in a stable manner, without causing large oscillations in the resource allocations. We achieve these goals by finding the value of $\bar{\mathbf{u}}_\mathbf{a}$ that minimizes the following cost function:

$$J_a = (\hat{y}_a(k) - 1)^2 + q\|\bar{\mathbf{u}}_\mathbf{a}(k) - \mathbf{u_a}(k-1)\|^2. \tag{2}$$

To explain the intuition behind this function, we define $J_p = (\hat{y}_a(k) - 1)^2$, and $J_c = \|\bar{\mathbf{u}}_\mathbf{a}(k) - \mathbf{u_a}(k-1)\|^2$. It is easy to see that J_p is 0 when $\hat{y}_a(k) = 1$ (or $y_a(k) = \bar{y}_a$), i.e., when application a is meeting its performance target. Otherwise, J_p serves as a penalty for the deviation of the application's measured performance from its target. Therefore, we refer to J_p as the *performance cost*.

The second function J_c, referred to as the *control cost*, is included to improve controller stability. The value of J_c is higher when the controller makes a larger change in the resource allocation in a single interval. Because $J_a = J_p + q \cdot J_c$, our controller aims to minimize a linear combination of both the performance cost and the control cost. Using the approximate linear relationship between the normalized performance and the resource allocations, as described by Eq. (1), we can derive the optimal resource allocations that minimize the cost function J_a, in terms of the recent resource allocations $\mathbf{u_a}(k-1)$ and the recent normalized performance values $\hat{y}_a$:

$$\begin{aligned} \bar{\mathbf{u}}^*_\mathbf{a}(k) = (\mathbf{b_0}\mathbf{b_0}^T + qI)^{-1}((1 - a_1\,\hat{y}_a(k-1) \\ -a_2\,\hat{y}_a(k-2) - \mathbf{b_1}^T\mathbf{u_a}(k-1))\mathbf{b_0} + q\mathbf{u_a}(k-1)), \end{aligned} \tag{3}$$

where I is an identity matrix. Note that the dependency of the model parameters a_1, a_2, $\mathbf{b_0}$ and $\mathbf{b_1}$ on the control interval k has been dropped from the equation to improve its readability.

To understand the intuition behind this control law and the effect of the scaling factor q, we define $\Delta\hat{y}_a(k) = 1 - a_1\,\hat{y}_a(k-1) - a_2\,\hat{y}_a(k-2) - \mathbf{b_1}^T\mathbf{u_a}(k-1)$. This indicates the discrepancy between the model-predicted value for $\hat{y}_a(k)$ and its target (which is 1) that needs to be compensated by the next allocation ($\mathbf{u_a}(k)$). For a small q value, $\bar{\mathbf{u}}^*_\mathbf{a}(k)$ is dominated by the effect of this discrepancy, and the controller reacts aggressively to reduce it. As the value of q increases, $\bar{\mathbf{u}}^*_\mathbf{a}(k)$ is increasingly dominated by the previous allocation ($\mathbf{u_a}(k-1)$), and the controller responds more slowly to the performance tracking error with less oscillation in the resulting resource allocations. In the extreme of an infinitely large q value, we have $\bar{\mathbf{u}}^*_\mathbf{a}(k) = \mathbf{u_a}(k-1)$, which means the allocations remain constant. As a result, the scaling factor q provides us an intuitive way to control the trade-off between the controller's stability and its ability to respond to changes in the workloads and performance, and hence is referred to as the *stability factor*.

3.2 Design of NodeController

For each of the virtualized nodes, a NodeController determines the allocation of resources to the applications, based on the resources requested by the AppControllers and the resources available at the node. This is required because the AppControllers act independently of one another and may, in aggregate, request more resources than the node has available. The NodeController divides the resources between the applications as follows. For each resource where the total allocation requested is less than the available capacity, the NodeController divides the resource in proportion to the requests from the AppControllers. This implies that not only will each requested allocation for this resource be satisfied, but the excess capacity will also be allocated in proportion to the requests. This allows the applications to achieve better performance than their targets when additional resources are available. For a resource that is contested, that is, where the sum of the resource requests is greater than the available capacity, the NodeController picks an allocation that locally minimizes the discrepancy (or *error*) between the resulting normalized application performance and its target value. More precisely, the cost function used is the weighted sum of the squared errors for the normalized performance across all applications sharing the node, where each application's weight represents its priority relative to other applications.

To illustrate this resource allocation method, let us take node1 in Figures 1 and 3 as an example (denoted as "n1"). This node is being used to host application 2 and the web tier of application 1. Suppose CPU and disk are the two critical resources being shared by the two applications. Then, the resource request from application 1 consists of two elements, $\bar{u}_{1,cpu,web}$ and $\bar{u}_{1,disk,web}$, one for each resource. Similarly, the resource request from application 2 consists of $\bar{u}_{2,cpu}$ and $\bar{u}_{2,disk}$. Because resource allocation is defined as a fraction of the total shared capacity of a resource, the resource requests from both applications need to satisfy the following capacity constraints:

$$\bar{u}_{1,cpu,web} + \bar{u}_{2,cpu} \leq 1 \tag{4}$$
$$\bar{u}_{1,disk,web} + \bar{u}_{2,disk} \leq 1 \tag{5}$$

When constraint (4) is violated, we say the virtualized node suffers *CPU contention*. Similarly, *disk contention* refers to the condition of the node when constraint (5) is violated. Next, we describe the two possible resource contention scenarios for the virtualized node n1, and the NodeController algorithm for dealing with each scenario.

3.2.1 Scenario I: Single resource contention

In this scenario, node n1 has enough capacity to meet the requests for one resource from the AppControllers, but not enough for the other resource; that is, one of constraints (4) and (5) is violated while the other is satisfied. The NodeController divides the resource that is enough in proportion to the requests. For the other resource $r \in R = \{cpu, disk\}$, both applications will receive less allocations than requested; let us denote the deficiencies as $\Delta u_{1,r,web} = \bar{u}_{1,r,web} - u_{1,r,web}$ and $\Delta u_{2,r} = \bar{u}_{2,r} - u_{2,r}$. The resulting discrepancy between the achieved and target normalized performance of application 1 can then be estimated as $\frac{\partial \hat{y}_1}{\partial u_{1,r,web}} \Delta u_{1,r,web}$. The penalty function for not meeting the performance targets is defined as:

$$J_{n1} = w_1 \left(\frac{\partial \hat{y}_1}{\partial u_{1,r,web}} \Delta u_{1,r,web} \right)^2 + w_2 \left(\frac{\partial \hat{y}_2}{\partial u_{2,r}} \Delta u_{2,r} \right)^2$$

The actual allocations are found by solving the following problem.

$$\begin{aligned} \text{Minimize } J_{n1} & \quad \text{subject to} & \\ \Delta u_{1,r,web} + \Delta u_{2,r} & \geq \bar{u}_{1,r,web} + \bar{u}_{2,r} - 1, & (6) \\ \Delta u_{1,r,web} & \geq 0, & (7) \\ \Delta u_{2,r} & \geq 0. & (8) \end{aligned}$$

Constraint (6) is simply the capacity constraint (4) or (5), applied to actual allocations. Constraints (7) and (8) ensure that no application is throttled to increase the performance of another application beyond its target. In the objective function J_{n1}, the discrepancies for the applications are weighted by their priority weights, so that higher priority applications experience less performance degradation.

From Eq. (1), we know that $\frac{\partial \hat{y}_1}{\partial u_{1,r,web}} = b_{0,1,r,web}$, and $\frac{\partial \hat{y}_2}{\partial u_{2,r}} = b_{0,2,r}$. Both coefficients can be obtained from the model estimators in the AppControllers for the two applications. This optimization problem is convex and a closed-form solution exists for the case of two applications sharing the node. For more than two applications, we use an off-the-shelf quadratic programming solver to compute the solution.

3.2.2 Scenario II: CPU and disk contention

This is the scenario where both CPU and disk are under contention. In this case, the actual allocations of CPU and disk for both applications will be below the respective requested amounts. The penalty function for not meeting the performance targets becomes:

$$\begin{aligned} J_{n1} &= w_1 \left(\sum_{r \in R} \frac{\partial \hat{y}_1}{\partial u_{1,r,web}} \Delta u_{1,r,web} \right)^2 \\ &\quad + w_2 \left(\sum_{r \in R} \frac{\partial \hat{y}_2}{\partial u_{2,r}} \Delta u_{2,r} \right)^2. \end{aligned}$$

The NodeController determines the actual allocations by minimizing J_{n1}, and by satisfying the constraints (6), (7), and (8) for both resources. This requires solving a convex optimization problem with the number of variables being the number of resource types multiplied by the number of VMs on the node. We have observed that the time taken for the online optimization is negligible (on average 40ms in our implementation).

4. Experimental Testbed

We built two testbeds using Xen to evaluate *AutoControl*. Each virtualized node in our testbeds hosts multiple applications. We used separate nodes to run the clients for these applications. The first testbed consisted of HP C-class blades, each equipped with two dual-core 2.2 GHz 64-bit processors with 4GB memory, two Gigabit Ethernet cards, and two 146 GB disks. We used OpenSuSE 10.3 as the OS and the default Xen (v3.1 built with 2.6.22.5-31 SMP kernel) in the OpenSuSE distribution. We also used OpenSuSE 10.3 to build VM images.

A second, larger testbed was built on Emulab [White 2002] to evaluate the scalability of *AutoControl*. We used the `pc3000` nodes on Emulab for server nodes, which contain a single Xeon 3GHz 64-bit processor (hyper-threading enabled) with 2GB memory, six Gigabit Ethernet cards (only two were used for our experiments) and two 146GB disks. The `pc850` nodes were used for client nodes. More details about the hardware can be found at the Emulab Web site. We used Fedora core 8 as the OS and a compiled version of latest Xen 3.3. We also used Fedora core 8 for the VM images.

In both testbeds, all of the VMs on a node shared one network interface and a disk; the remaining resources were used by Dom0 running the controllers and a monitoring framework. This setup allowed us to experiment easily with CPU and disk bottlenecks in hosted applications. The monitoring framework (which implements the sensors for the controllers) periodically collects two types of statistics: resource utilization and application performance. We used Xen's `xm` command to collect CPU utilization and `iostat` command to collect disk usage statistics. We instrumented the applications so that we can directly collect performance statistics. *AutoControl* uses two mechanisms to enforce the resource allocations: Xen's credit-based CPU scheduler and an interposed proportional-share I/O scheduler [Gulati 2007] that we implemented. *AutoControl* interacts with the schedulers by assigning a *cap* for each VM's CPU usage and a *disk share* for each VM's disk usage in every control interval.

We used three different applications in our experiments: RUBiS [Amza 2002], an online auction site benchmark; a Java implementation of the TPC-W benchmark [Cain 2001]; and a custom-built secure media server. RUBiS and TPC-W use a multi-tier setup consisting of web and database tiers. They both provide workloads of different mixes. For RUBiS, we used a workload mix called the browsing mix that simulates users browsing through an auction site. For TPC-W, we used the shopping mix, which simulates users interacting with a shopping site. The browsing mix stresses the web tier, while the shopping mix exerts more demand on the database tier.

The secure media (smedia) server is a representation of a media server that can serve encrypted media streams. The smedia server runs a certain number of concurrent threads, each serving a client that continuously requests media files from the server. A media client can request an encrypted or unencrypted stream. Upon receiving the request, the server reads the particular media file from the disk (or from memory if it is cached), optionally encrypts it, and sends it to the client. Reading a file from the disk consumes disk I/O resource, and encryption requires CPU resource. For a given number of threads, by changing the fraction of the client requests for encrypted media, we can vary the amount of CPU or disk I/O resource used. This flexibility allowed us to study our controller's behavior for CPU and disk I/O bottlenecks.

All three applications use a closed-loop client model where a new request is only issued after the previous request is complete. We have instrumented the client workload generators for these applications such that the number of concurrent clients can be dynamically adjusted at runtime to vary the level of stress imposed on the hosting nodes.

To test whether *AutoControl* can handle the dynamic variations in resource demands seen by typical enterprise applications, we also used resource utilization traces from an SAP application server running in a production environment. We dynamically varied the number of concurrent threads for RUBiS, TPC-W or smedia to recreate the resource consumption of these workloads on our test nodes. For example, to create 40% average CPU utilization over a 5 minute period, we used 500 threads simulating 500 concurrent users. Note that we only matched the CPU utilization of the production trace. We did not attempt to recreate the disk utilization, because the traces did not contain the needed metadata.

We also used traces generated from a media workload generator called MediSyn [Tang 2003]. MediSyn generates traces that are based on analytical models drawn from real-world traces collected at an HP Labs production media server. It captures important properties of streaming media workloads, including file duration, popularity, encoding bit rate, and streaming session time. We re-created the access pattern of the trace by closely following the start times, end times, and bitrates of the sessions. We did not attempt to re-create the disk access pattern, because of the lack of metadata. A more detailed description of recreating the production traces is included in [Padala 2008].

5. Evaluation Results

In this section, we present our experimental results for *AutoControl*. These experiments were designed to test the following capabilities of our system:

1. Automatically detect and mitigate dynamically-changing resource bottlenecks across application tiers;
2. Enforce performance targets for metrics including throughput and response time under dynamically-varying resource demands;
3. Prioritize among applications during resource contention;
4. Scale to a reasonably large testbed.

5.1 Detecting and mitigating resource bottlenecks

Our first experiment evaluates *AutoControl* when resource bottlenecks occur dynamically and shift from one resource to another or from one tier to the next. For this experiment,

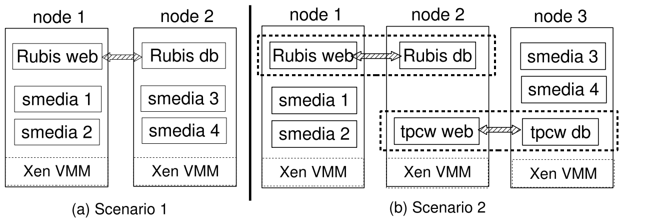


Figure 5. Experimental setup

Table 2. Percentage of encrypted streams in each smedia application in different time intervals

Intervals	smedia1	smedia2	smedia3	smedia4
1-29	50%	50%	2%	2%
30-59	2%	2%	2%	2%
60-89	2%	2%	50%	50%

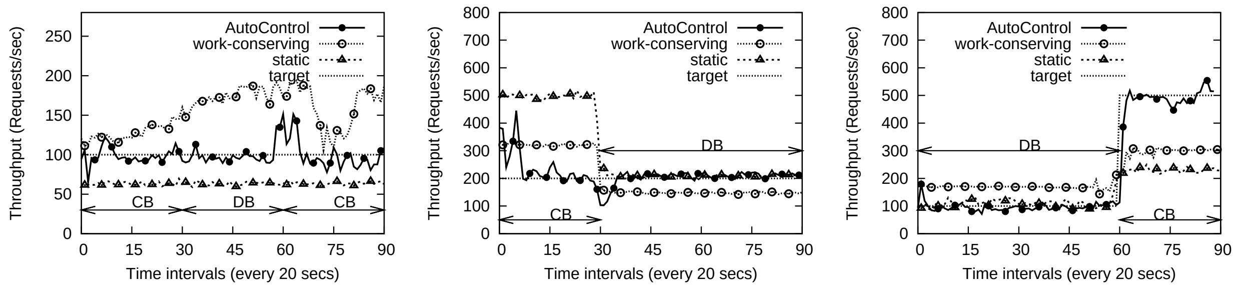
we used the setup shown in Figure 5, where two physical nodes host four smedia applications and a RUBiS application spanning both nodes. This small setup allows us to gain insight into the system behavior and to validate the internal working of the model estimator and the MIMO controller.

In this experiment, we varied the percentage of encrypted streams requested by the smedia clients over time to create resource bottleneck shifts in each of the virtualized nodes (Table 2). Throughout the experiment, smedia1 and smedia3 had throughput targets of 200 reqs/sec each. We alternated the throughput targets of smedia2 and smedia4 between 500 reqs/sec and 100 reqs/sec. The targets were chosen such that both nodes were running near their capacity limits for either CPU or disk I/O. During the CPU-heavy phase, a 10KB file was fetched, while during the disk-heavy phase, an 80KB file was fetched.

5.1.1 Results under *AutoControl*

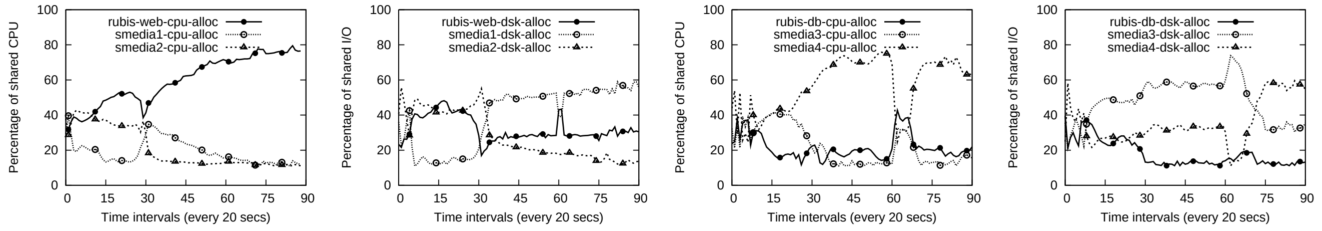
Figure 6 shows the throughput of RUBIS, smedia1, and smedia4 in each control interval of 20 seconds. (Smedia2 and smedia3 behaved similarly and are not shown for lack of space.) The labels "CB" and "DB" indicate when CPU or disk I/O was a bottleneck and when the bottleneck shifted from one type of resource to another. To help understand how *AutoControl* achieved the performance targets, Figure 7 shows the CPU and disk I/O allocations to all five applications (6 VMs) on both nodes.

For the first 29 intervals, the RUBiS web tier, smedia1 and smedia2 contended for CPU on node 1. From Figures 7(a) and 7(b), we see that the controller gave different portions of both CPU and disk resources to the three VMs on node 1 such that all of their targets could be met. In the same time period, on node 2, the RUBiS database tier, smedia3 and smedia4 were contending for the disk I/O. Figures 7(c) and 7(d) show the CPU and disk I/O allocations for the three VMs on this node. The controller not only allocated the right proportion of disk I/O to smedia3 and smedia4 for them to achieve their throughput targets, it also allocated the right amount of CPU to the RUBiS database tier so that the two-tier application could meet its target.



(a) RUBiS throughput (b) Smedia1 throughput (c) Smedia4 throughput

Figure 6. Throughput of different applications with bottlenecks in CPU or disk I/O and across multiple nodes. The time periods with a CPU bottleneck are labeled as "CB", and those with a disk bottleneck are labeled as "DB."



(a) CPU allocations on node 1 (b) Disk allocations on node 1 (c) CPU allocations on node 2 (d) Disk allocations on node 2

Figure 7. Resource allocations to different applications or application tiers on different nodes

At interval 30, the workloads for the smedia applications on node 1 were switched to be disk-heavy. As a result, smedia1 and smedia2 were contending for disk I/O, since the RUBiS web tier used minimal disk resource. The controller recognized this change in resource bottleneck automatically and ensured that both smedia1 and smedia2 could meet their new throughput targets by allocating the right amount of disk resources to both smedia applications (see Figure 7(b)).

At interval 60, the workloads for the smedia applications on node 2 were switched to be CPU-heavy. Because the RUBiS database tier also required a non-negligible amount of CPU (around 20%), smedia3 and smedia4 started contending for CPU with the RUBiS database tier on node 2. Again, the controller was able to automatically translate the application-level goals into appropriate resource allocations to the three VMs on node 2 (see Figure 7(c)).

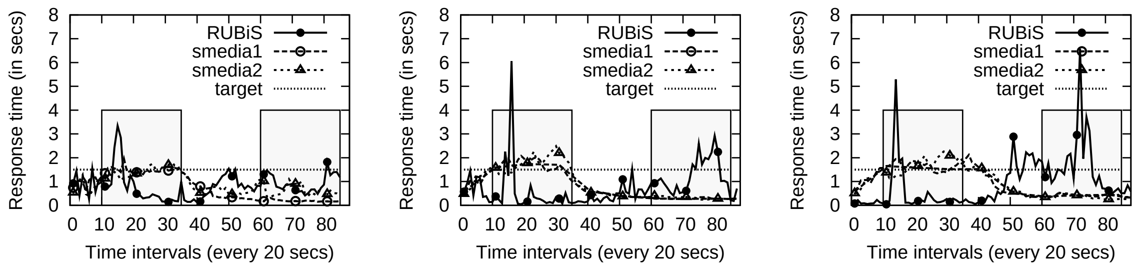
These results show that *AutoControl* was able to achieve the performance targets for all the applications even though (i) the resource bottleneck occurred either in the CPU or in the disk or shifted from one to the other; and (ii) both tiers of the RUBiS application, distributed across two physical nodes, experienced resource contention.

5.1.2 Comparison with the state-of-the-art

For comparison, we repeated the same experiment using two resource allocation methods that are commonly used on consolidated infrastructures today: a work-conserving mode and a static allocation mode. In the work-conserving mode, the applications run in the default Xen settings, where a cap of zero is specified for the shared CPU on a node, indicating that the applications can use any amount of CPU resources. In this mode, our proportional share disk scheduler was unloaded to allow unhindered disk access. In the static mode, the three applications sharing a node were allocated CPU and disk resources in the fixed ratio of 20:50:30. The resulting application performance from both approaches is shown in Figure 6 along with the performance from *AutoControl*. As can be seen, neither approach was able to offer the degree of performance assurance provided by *AutoControl*.

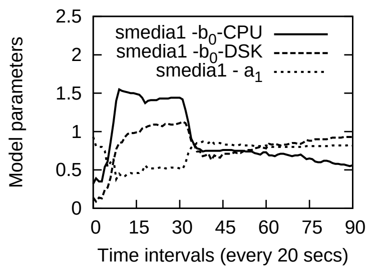
For the work-conserving mode, RUBiS was able to achieve a throughput much higher than its target at the cost of performance degradation in the other applications sharing the same infrastructure. The remaining capacity on either node was equally shared by smedia1 and smedia2 on node 1, and smedia3 and smedia4 on node 2. This mode did not provide service differentiation between the applications according to their respective performance targets.

For the static mode, RUBiS was never able to reach its performance target given the fixed allocation, and the smedia applications exceeded their targets at some times and missed the targets at other times. Given the changes in workload behavior for the smedia applications, there is no fixed allocation ratio for either CPU or disk I/O that will guarantee the performance targets for all the applications.

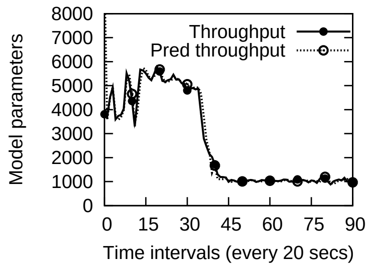


(a) *AutoControl* (b) Work-conserving mode (c) Static allocation mode

Figure 8. Performance comparison of *AutoControl*, work-conserving mode and static allocation mode, while running RUBiS, smedia1, and smedia2 with production-trace-driven workloads.



(a) Model parameter values for smedia1



(b) Measured and model-predicted throughput for smedia2

Figure 9. Internal workings of the *AppController* - model estimator performance

5.1.3 Evaluating the model estimator

To understand further the internal workings of *AutoControl*, we now demonstrate a key element of our design - the model estimator in the *AppController* that automatically determines the dynamic relationship between an application's performance and its resource allocations. Our online estimator continuously adapts the model parameters as dynamic changes occur in the system. Figure 9(a) shows the model parameters ($b_{0,cpu}$, $b_{0,disk}$, and a_1) for smedia1 as functions of the control interval. For lack of space, we omit the second-order parameters and the parameter values for the other applications. As we can see, the values of $b_{0,cpu}$, representing the correlation between application performance and CPU allocation, dominated the $b_{0,disk}$, and a_1 parameters for the first 29 intervals. The disk allocation also mattered, but was not as critical. This is consistent with our observation that node 1 had a CPU bottleneck during that period. After the 30th interval, when disk became a bottleneck on node 1, while CPU became less loaded, the model coefficient $b_{0,disk}$ exceeded $b_{0,cpu}$ and became dominant after a period of adaptation.

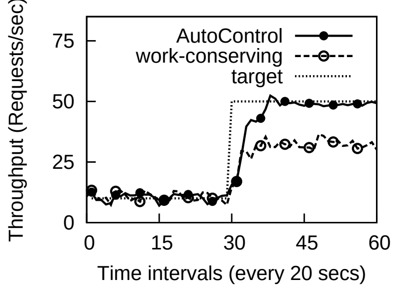
Table 3. Two prediction accuracy measures of linear models (in percentage)

	rubis	smedia1	smedia2	smedia3	smedia4
R^2	79.8	91.6	92.2	93.3	97.0
MAPE	4.2	5.0	6.9	4.5	8.5

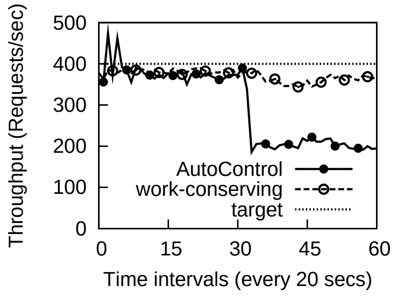
To assess the overall prediction accuracy of the linear models, we computed two measures, the coefficient of determination (R^2) and the mean absolute percentage error (MAPE), for each application. R^2 and MAPE can be calculated as $R^2 = 1 - \frac{\sum_{k=1}(\hat{y}_a^p(k) - \hat{y}_a(k))^2}{\sum_k(\hat{y}_a^p(k) - \hat{y}_{a,avg})^2}$, and $MAPE = \frac{1}{K}\sum_{k=1}^{K} |\frac{\hat{y}_a^p(k) - \hat{y}_a(k)}{\hat{y}_a(k)}|$, where K is the total number of samples, $\hat{y}_a^p(k)$ and $\hat{y}_a(k)$ are the model-predicted value and the measured value for the normalized performance of application a, and $\hat{y}_{a,avg}$ is the sample mean of $\hat{y}_a$. Table 3 shows the values of these two measures for all the five applications. As an example, we also show in Figure 9(b) the measured and the model-predicted throughput for smedia2. From both the table and the figure, we can see that our model is able to predict the normalized application performance accurately, with R^2 above 80% and MAPE below 10%. This validates our belief that low-order linear models, when adapted online, can be good enough local approximations of the system dynamics even though the latter is nonlinear and time-varying.

5.2 Handling dynamic demands and nonlinearity

In this section, we evaluate whether *AutoControl* can meet application performance targets under dynamic variations in resource demands. For this experiment, we continued to use the setup in Figure 5(a), while driving each of the RUBiS, smedia1 and smedia2 applications using CPU utilization traces from a production site, using the method described in Section 4. Furthermore, we chose response time as the performance metric. Response times behave nonlinearly with

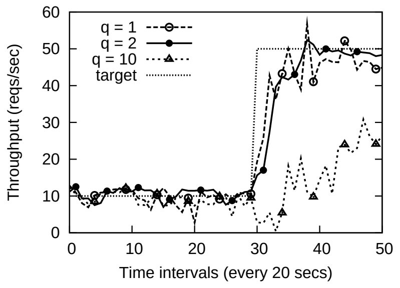


(a) TPC-W throughput

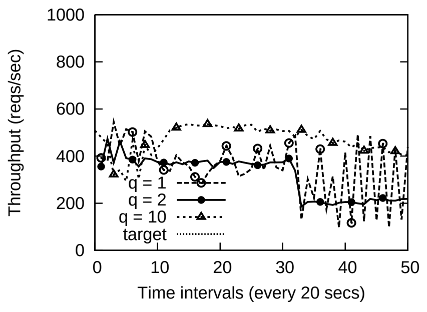


(b) Smedia3 throughput

Figure 10. Performance comparison between *AutoControl* and work-conserving mode, with different priority weights for TPC-W ($w = 2$) and smedia3 ($w = 1$).



(a) TPC-W throughput



(b) Smedia3 throughput

Figure 11. Performance results for TPC-W and smedia3 with stability factor $q = 1, 2, 10$

respect to resource allocations and can be used to evaluate how *AutoControl* copes with nonlinearity in the system.

For smedia3 and smedia4 on node 2, we ran a workload with 40 threads with a 2% chance of requesting an encrypted stream (making it disk-bound). For brevity, we only show the results for the three applications running on node 1. Figures 8(a), 8(b) and 8(c) show the measured average response times of RUBiS, smedia1 and smedia2 as functions of the control interval, using *AutoControl*, work-conserving mode, and static allocation mode, respectively. We used a response time target of 1.5 second for all the three applications, and set the CPU allocation at a fixed 33% for each application in the static mode. The dark-shaded regions show the time intervals when a CPU bottleneck occurred.

In the first region, for the work-conserving mode, both smedia1 and smedia2 had high CPU demands, causing not only response time target violations for themselves, but also a large spike of 6 second in the response time for RUBiS at the 15th interval. In comparison, *AutoControl* allocated higher shares of the CPU to both smedia1 and smedia2 without overly penalizing RUBiS. As a result, all the three applications were able to meet the response time target most of the time, except for the small spike in RUBiS.

In the second shaded region, the RUBiS application became more CPU-intensive. Because there is no performance assurance in the work-conserving mode, the response time of RUBiS surged and resulted in a period of target violations, while both smedia1 and smedia2 had response times well below the target. In contrast, *AutoControl* allocated more CPU capacity to RUBiS when needed by carefully reducing the resource allocation to smedia2. The result was that there were almost no target violations for the three applications.

The result from the static allocation mode was similar to that from the work-conserving mode, except that the RUBiS response time was even worse in the second region.

Despite the fact that response time is a nonlinear function of resource allocations, and that the resource demands taken from the real traces were much more dynamic, *AutoControl* was still able to balance the resources and minimize the response time violations for all three applications.

5.3 Enforcing application priorities

In this section, we evaluate *AutoControl* during periods of resource contention where there are not enough resources to meet all of the application targets. We used the experimental setup shown in Figure 5(b) with two multi-tier applications, RUBiS and TPC-W, and four smedia applications spanning three nodes. We drove TPC-W with the shopping mix workload with 200 concurrent threads; and used the workloads in the first experiment for RUBIS and smedia instances. We assume that the TPC-W application is of higher priority than the two smedia applications sharing the same node. Therefore, TPC-W was assigned a priority weight of $w = 2$ while the other applications had $w = 1$ in order to provide service differentiation.

Unlike the setup used in previous experiments, there was no resource contention on node 2. For the first 29 intervals, all six applications were able to meet their goals. Figure 10 shows the throughput targets and the achieved throughput for TPC-W and smedia3. (The other four applications are not shown to save space.) At interval 30, 800 more threads were added to the TPC-W client, emulating increased user activity. The throughput target for TPC-W was adjusted from 20 to 50 requests/sec to reflect this change. This increased the CPU load on the database tier creating a CPU bottleneck on node 3. *AutoControl* responded to this change automatically and correctly re-distributed the resources. Note that not all three applications (TPC-W, smedia3, and smedia4) on node 3 could reach their targets. However the higher priority weight for TPC-W allowed it to still meet its throughput target while degrading performance for the other two applications.

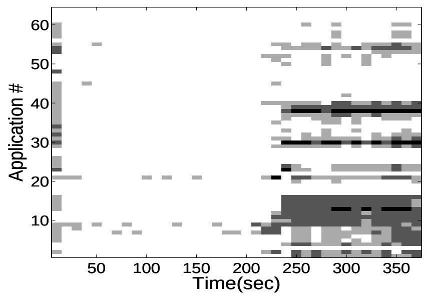
The result from using the work-conserving mode for the same scenario is also shown in Figure 10. In this mode, smedia3 and smedia4 took up more CPU resource, causing the high-priority TPC-W application to fall below its target.

We also use this example to illustrate how a tradeoff between controller stability and responsiveness can be handled by adjusting the stability factor q. Figure 11 shows the achieved throughput for TPC-W and smedia3 under the same workload condition, for q values of 1, 2, and 10. The result for $q = 2$ is the same as in Figure 10. For $q = 1$, the controller reacts to the workload change more quickly and aggressively, resulting in large oscillations in performance. For $q = 10$, the controller becomes much more sluggish and does not adjust resource allocations fast enough to track the performance targets.

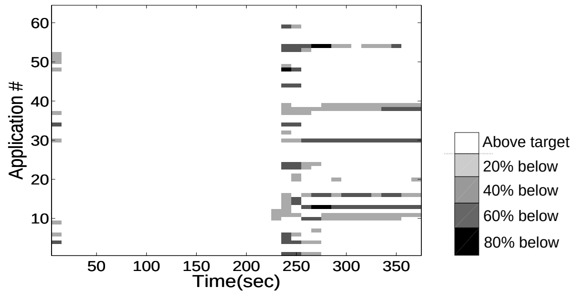
5.4 Scalability experiments

In this section, we evaluate the scalability of *AutoControl* using a larger testbed built on Emulab [White 2002], using 16 server nodes, each running 4 smedia applications in 4 individual virtual machines. An additional 16 client nodes running 64 clients were used to generate the workloads for the 64 smedia servers. Initially, each application used a light workload keeping all the nodes underloaded. After 240 seconds, we increased the load for half of the applications (in 32 VMs) and updated their performance targets accordingly. These applications were chosen randomly and hence were not spread uniformly across all the nodes. The numbers of nodes that had 0, 1, 2, and 3 applications with increased load were 1, 4, 5, and 6, respectively.

Figure 12(a) and 12(b) show the SLO violations of the 64 applications over time, using the work-conserving mode and *AutoControl*, respectively. The x-axis shows the time in seconds and the y-axis shows the application index. The gray shades in the legend represent different levels of SLO violations (the darker the worse), whereas the white color indicates no SLO violations.



(a) Work-conserving mode



(b) *AutoControl*

Figure 12. SLO violations in 64 applications using work-conserving mode and *AutoControl*

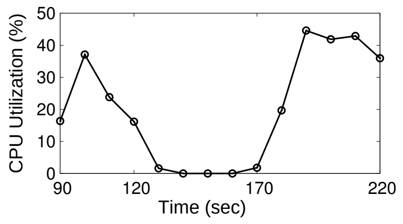


Figure 13. Measurement of CPU utilization on a loaded VM during migration

The nodes that had no or only one application with a heavy load remained underloaded and there were almost no SLO violations. When there were two applications with increased load on a single node, the node was slightly overloaded and the work-conserving mode resulted in SLO violations in the applications sharing the node, whereas *AutoControl* was able to re-distribute the resources and significantly reduce the SLO violations. However, if a node had three applications with increased load, even *AutoControl* was not able to avoid SLO violations for certain applications because no re-distribution policy could satisfy the resource demands of all the applications.

6. Discussion and Future Work

This section describes some of the design issues in *AutoControl*, alternative methods and future research work.

6.1 Migration for dealing with bottlenecks

AutoControl enables dynamic re-distribution of resources between competing applications to meet their targets, so long as sufficient resources are available. If a node is persistently overloaded, VM migration [Clark 2005, Wood 2007] may be needed, but the overhead of migration can cause additional SLO violations. We performed experiments to quantify these violations. Migrating a lightly-loaded smedia server hosted in a 512MB VM takes an average of 6.3 seconds. However, during the migration, we observed as much as 79% degradation of smedia throughput and CPU utilization as high as 94% on the source node. Since migration usually takes place when a node is heavily loaded, we also measured the overhead of migration under various overload conditions. We have used two or more smedia VMs with varying degrees of overload, and found the migration times vary between 13 to 50 seconds. Figure 13 shows the CPU utilization of one such migrating VM. There were four heavily-loaded smedia VMs on the source node and one lightly-loaded VM on the destination node in this setup. During the migration period (t=120-170), the VM showed CPU starvation, and we observed a significant decrease in smedia performance. We plan to extend *AutoControl* to include VM migration as an additional mechanism, and expect that a combination of VM migration and *AutoControl* will provide better overall performance than using one or the other.

6.2 Actuator & sensor behavior

The behavior of sensors and actuators affects our control. In existing systems, most sensors return accurate information, but many actuators are poorly designed. We observed various inaccuracies with Xen's earlier SEDF scheduler and credit scheduler that are identified by other researchers [Gupta 2006] as well. These inaccuracies cause VMs to gain more or less CPU than set by the controller. Empirical evidence shows that our controller is resistant to CPU scheduler's inaccuracies.

6.3 Network and memory control

We are also working on extending *AutoControl* for network and memory resources. Our initial experiments to incorporate network control using Linux's traffic controller (`tc`) found its overhead to be prohibitive. We are currently investigating other mechanisms for network control.

VMware ESX Server allows memory overbooking and dynamically re-allocates memory from one virtual machine to another through the use of balloon drivers [Waldspurger 2002]. Using Xen's balloon driver, we have developed policies for automated memory control [Heo 2009] and will soon integrate them with *AutoControl*.

7. Related Work

In recent years, control theory has been applied to computer systems for resource management and performance control [Hellerstein 2004, Karamanolis 2005]. Examples of its application include web server performance guarantees [Abdelzaher 2002], dynamic adjustment of the cache size for multiple request classes [Lu 2004], CPU and memory utilization control in web servers [Diao 2002], adjustment of resource demands of virtual machines based on resource availability [Zhang 2005], and dynamic CPU allocations for multi-tier applications [Liu 2007, Padala 2007]. These concerned themselves with controlling only a single resource (usually CPU), used mostly single-input single-output (SISO) controllers (except in [Diao 2002]), and required changes in the applications. In contrast, our MIMO controller operates on multiple resources (CPU and storage) and uses the sensors and actuators at the virtualization layer and external QoS sensors without requiring any modifications to applications.

In [Diao 2002], the authors apply MIMO control to adjust two configuration parameters within Apache to regulate CPU and memory utilization of the Web server. They used static linear models, which are obtained by system identification for modeling the system. Our earlier attempts at static models for controlling CPU and disk resources have failed, and therefore, we used a dynamic adaptive model in this paper. Our work also extends MIMO control to controlling multiple resources and virtualization, which has more complex interactions than controlling a single web server.

Prior work on controlling storage resources independent of CPU includes systems that provide performance guarantees in storage systems [Chambliss 2003, Jin 2004, Lumb 2003]. However, one has to tune these tools to achieve application-level guarantees. Our work builds on top of our earlier work, where we developed an adaptive controller [Karlsson 2004] to achieve performance differentiation, and an efficient adaptive proportional share scheduler [Gulati 2007] for storage systems.

Traditional admission control to prevent computing systems from being overloaded has focused mostly on web servers. Control theory was applied in [Kamra 2004] for the design of a self-tuning admission controller for 3-tier web sites. In [Karlsson 2004], a self-tuning adaptive controller was 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 our approach, because the former shapes the resource demand into a server system, whereas the latter adjusts the supply of resources for handling the demand.

Resource management has been widely studied and large body of the work falls into providing mechanisms for resource management. In [Banga 1999], resource containers are proposed to achieve fine-grained resource management. The container approach has been extended to full scale virtualization by VMware [Rosenblum 1999], Xen [Barham 2003] and others. These technologies provide mechanisms to allocate fine-grained resources and our work uses these mechanisms to set the policies to achieve application goals.

Proportional share schedulers allow reserving CPU capacity for applications [Jones 1997, Nieh 1997, Waldspurger 1994]. 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 [Steere 1999] 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 application performance that spans multiple tiers in a given application.

Dynamic resource allocation in distributed systems has been studied extensively, but the emphasis has been on allocating resources across multiple nodes rather than in time, because of lack of good isolation mechanisms like virtualization. It was formulated as an online optimization problem in [Aron 2000] using periodic utilization measurements, and resource allocation was implemented via request distribution. Resource provisioning for large clusters hosting multiple services was modeled as a "bidding" process in order to save energy in [Chase 2001]. The active server set of each service was dynamically resized adapting to the offered load. In [Shen 2002], an integrated framework was proposed combining a cluster-level load balancer and a node-level class-aware scheduler to achieve both overall system efficiency and individual response time goals. However, these existing techniques are not directly applicable to allocating resources to applications running in VMs. They also fall short of providing a way of allocating resources to meet the end-to-end SLOs.

Profiling and predicting resource usage in consolidated environments has been studied in [Urgaonkar 2002, Wood 2008, Choi 2008]. These studies are complementary to our work, and can help create a better model for resource usage of applications that can be used to drive our controller.

8. Conclusions

In this paper, we presented *AutoControl*, a feedback control system to dynamically allocate resources to applications running on shared virtualized infrastructure. It consists of an online model estimator that captures the dynamic relationship between application-level performance and resource allocations and a MIMO resource controller that determines appropriate allocations of multiple resources to achieve application-level SLOs.

We evaluated *AutoControl* using two testbeds consisting of varying numbers of Xen virtual machines and various single- and multi- tier applications and benchmarks. Our experimental results confirmed that *AutoControl* can detect dynamically-changing CPU and disk bottlenecks across multiple nodes and can adjust resource allocations accordingly to achieve end-to-end application-level SLOs. In addition, *AutoControl* can cope with dynamically-shifting resource bottlenecks and provide a level of service differentiation according to the priorities of individual applications. Finally, we showed that *AutoControl* can enforce performance targets for different application-level metrics, including throughput and response time, under dynamically-varying resource demands.

9. Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Galen Hunt. This project was supported in part by NSF grant CCF-0444417.

References

[Abdelzaher 2002] 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(1), 2002.

[Amza 2002] C. Amza, A. Chanda, 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 the 5th IEEE Annual Workshop on Workload Characterization*, October 2002.

[Aron 2000] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of ACM SIGMETRICS*, pages 90–101, 2000.

[Astrom 1995] K.J. Astrom and B. Wittenmark. *Adaptive Control*. Addition-Wesley, 1995.

[Banga 1999] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–58, 1999.

[Barham 2003] 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 Symposium on Operating Systems Principles (SOSP)*, October 2003.

[Cain 2001] H.W. Cain, R. Rajwar, M. Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, 2001.

[Chambliss 2003] D.D. Chambliss, G.A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T.P. Lee. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS)*, October 2003.

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

[Choi 2008] J. Choi, S. Govindan, B. Urgaonkar, and A. Sivasubramaniam. Profiling, prediction, and capping of power consumption in consolidated environments. In Ethan L. Miller and Carey L. Williamson, editors, *MASCOTS*, pages 3–12. IEEE Computer Society, 2008.

[Clark 2005] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2005.

[Diao 2002] 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.

[Gulati 2007] A. Gulati, A. Merchant, M. Uysal, and P.J. Varman. Efficient and adaptive proportional share I/O scheduling. Technical Report HPL-2007-186, HP Labs, November 2007.

[Gupta 2006] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen, 2006.

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

[Heo 2009] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *Proceedings of IFIP/IEEE Symposium on Integrated Management (IM'09) mini-conference*, June 2009.

[Jin 2004] W. Jin, J.S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of ACM SIGMETRICS*, 2004.

[Jones 1997] 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 Symposium on Operating System Principles (SOSP)*, October 1997.

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

[Karamanolis 2005] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of HotOS*, June 2005.

[Karlsson 2004] 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.

[Liu 2007] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *Proceedings of IEEE Conference on Decision and Control (CDC)*, 2007.

[Lu 2004] 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.

[Lumb 2003] C.R. Lumb, A. Merchant, and G.A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of File and Storage Technologies (FAST)*. USENIX, 2003.

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

[Padala 2008] P. Padala, K. Hou, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. G. Shin. Automated control of multiple virtualized resources. Technical Report HPL-2008-123, HP Labs, Oct 2008.

[Padala 2007] P. Padala, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, K. Salem, and K. G. Shin. Adaptive control of virutalized resources in utility computing environments. In *Proceedings of EuroSys*, 2007.

[Rolia 2005] J. Rolia, L. Cherkasova, M. Arlit, and A. Andrzejak. A capacity management service for resource pools. In *Proceedings of International Workshop on Software and Performance*, July 2005.

[Rosenblum 1999] M. Rosenblum. VMware's Virtual Platform: A virtual machine monitor for commodity PCs. In *Hot Chips 11*, 1999.

[Shen 2002] 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.

[Steere 1999] 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.

[Tang 2003] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat. Long-term streaming media server workload analysis and modeling. Technical Report HPL-2003-23, HP Labs, February 07 2003.

[Urgaonkar 2002] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th Symposium on operating systems design and implementation (OSDI)*, pages 239 – 254, December 2002.

[Waldspurger 1994] 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.

[Waldspurger 2002] C.A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[White 2002] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, December 2002.

[Wood 2008] T. Wood, L. Cherkasova, K. M. Ozonat, and P. J. Shenoy. Profiling and modeling resource usage of virtualized applications. In Valérie Issarny and Richard E. Schantz, editors, *Middleware*, volume 5346 of *Lecture Notes in Computer Science*, pages 366–387. Springer, 2008.

[Wood 2007] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2007.

[Zhang 2005] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: Leveraging a feedback-control model for application adaptation. In *Proceedings of the Virtual Execution Environments (VEE)*, 2005.