Automated Control of Multiple Virtualized Resources

Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant

HP Laboratories
HPL-2008-123R1

Keyword(s):
virtualization, consolidation, service level objective, control, service differentiation

Abstract:
Virtualized data centers enable consolidation of multiple applications and sharing of multiple resources among these applications. However, current virtualization technologies are inadequate in achieving complex service level objectives (SLOs) for enterprise applications with time-varying demands for multiple resources. In this paper, we present *AutoControl*, a resource allocation system that automatically adapts to dynamic workload changes in a shared virtualized infrastructure 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 resources to ensure application SLOs. Our experimental results using RUBiS and TPC-W benchmarks along with production-trace-driven workloads indicate that *AutoControl* can detect and adapt to CPU and disk I/O bottlenecks that occur over time and across multiple nodes and allocate multiple virtualized resources accordingly to achieve application SLOs. It can also provide service differentiation according to the priorities of individual applications during resource contention.

External Posting Date: November 21, 2008 [Fulltext] - Approved for External Publication

Internal Posting Date: November 21, 2008 [Fulltext]

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, Mustafa Uysal,
Zhikui Wang, Sharad Singhal, Arif Merchant
Hewlett Packard Laboratories
{firstname.lastname}@hp.com

Abstract

Virtualized data centers enable consolidation of multiple applications and sharing of multiple resources among these applications. However, current virtualization technologies are inadequate in achieving complex service level objectives (SLOs) for enterprise applications with time-varying demands for multiple resources. In this paper, we present *AutoControl*, a resource allocation system that automatically adapts to dynamic workload changes in a shared virtualized infrastructure 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 resources to ensure application SLOs. Our experimental results using RUBiS and TPC-W benchmarks along with production-trace-driven workloads indicate that *AutoControl* can detect and adapt to CPU and disk I/O bottlenecks that occur over time and across multiple nodes and allocate multiple virtualized resources accordingly to achieve application SLOs. It can also provide service differentiation according to the priorities of individual applications during resource contention.

1. INTRODUCTION

Virtualization is causing a disruptive change in enterprise data centers and giving rise to a new paradigm: *shared virtualized infrastructure*. In this new paradigm, multiple enterprise applications share dynamically allocated resources. These applications are also consolidated to reduce infrastructure and operating costs while simultaneously increasing resource utilization. As a result, data center administrators are faced with growing challenges to meet service level objectives (SLOs) in the presence of dynamic resource sharing and unpredictable interactions across many applications. These challenges include:

- *Complex SLOs*: It is non-trivial to convert individual application SLOs to corresponding resource shares in the shared virtualized platform. For example, determining the amount of CPU and the disk shares required to achieve a specified number of financial transactions is difficult.

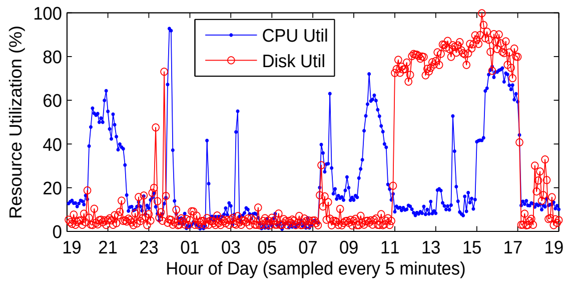


Figure 1: Resource usage in a production SAP application server for a one-day period.

- *Time-varying resource requirements*: The intensity and the mix of enterprise application workloads change over time. As a result, the demand for individual resource types changes over the lifetime of the application. For example, Figure 1 shows the CPU and disk utilization of an SAP application measured every 5 minutes during a 24-hour period. The utilization of both resources varies over time considerably, and the peak utilization of the two resources 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, wasting resources.

- *Distributed resource allocation*: 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 system-level resources.

Researchers have studied capacity planning for such an environment by using historical resource utilization traces to predict the application resource requirements in the future and to place compatible sets of applications onto the shared nodes [23]. Such an 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 forWeb-based, interactive applications. Furthermore, in a 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 [27]. 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 virtualized nodes and that 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 and capture the relationship between application level performance and the allocation of individual resource shares. Our adaptive modeling approach captures the complex behavior of enterprise applications including varying resource demands over time, resource demands from distributed application components, and shifting demands across multiple resources types. Second, we design a two-layered, multi-input, multi-output (MIMO) controller to *automatically* allocate multiple types of resources to multiple enterprise applications to achieve their SLOs. The first layer consists of a set of application controllers that automatically determines the amount of resources necessary to achieve individual application SLOs, using the estimated models and a feedback approach. The second layer is comprised of a set of node controllers that detect resource bottlenecks on the shared nodes and properly allocate resources of multiple types to individual applications. In overload cases, the node controllers can provide service differentiation by prioritizing allocations among different applications.

We have built a testbed using Xen [5], and evaluated our controller 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 multitier 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.

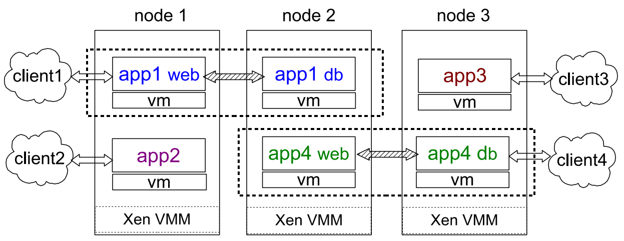


Figure 2: Physical organization: Each node hosts multiple applications running on VMs. Applications can span multiple nodes.

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 a discussion of related work in Section 6. Section 7 delineates some of the limitations of this work along with suggestions for future research, and conclusions are drawn in Section 8.

2. OVERVIEW, ASSUMPTIONS AND GOALS

In this section, we present an overview of our system architecture and the assumptions and goals that drive our design. We assume that applications are hosted within containers or virtual machines (VM) [5] to enable resource sharing within a virtualized server node. A multi-tier application may run on multiple VMs that span nodes. Figure 2 shows an example with three nodes hosting four applications.

In *AutoControl*, operators can specify the 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 two resources, and application throughput or average response time as the performance metric.

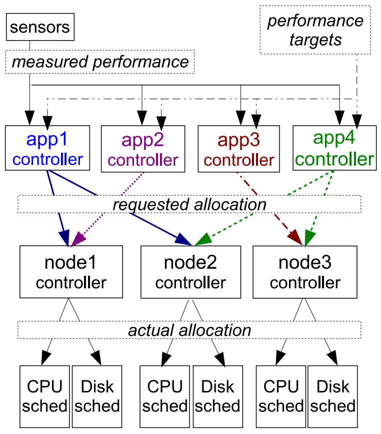


Figure 3: Logical controller organization: Each application has one application controller. Each node has one node controller that arbitrates the requests from multiple application controllers.

We set the following design goals for *AutoControl*:

Performance assurance: If all applications can meet their performance targets, *AutoControl* should allocate resources properly to achieve them. If they cannot be met, *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* without human intervention.

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; and also limit the number of variables each controller deals with.

Based on these principles, we have designed AutoControl with a two-layered, 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 2. 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, and based on the discrepancy, automatically determines the resource allocations needed for the next control interval, and sends these requests to the NodeControllers for the nodes that host the application.

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., $R = \{cpu, disk\}$
k	index for control interval
$x(k)$	value of variable x in control interval k
$ur_{a,r,t}$	requested allocation of resource type r to tier t of application a, $0 \leq u_{a,r,t}(k) \leq 1$ ($ur_{a,r}$ 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
yr_a	performance target for application a
yn_a	normalized performance for application a, where $yn_a = y_a/yr_a$
w_a	priority weight for application a
q	stability factor in the AppController

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 allocation 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 resources to the VMs in real time. Figure 3 shows CPU and disk schedulers as examples.

The *AutoControl* architecture allows the placement of AppControllers and NodeControllers in a distributed fashion. NodeControllers can be hosted in the physical node they are controlling. AppControllers can be hosted in a node where one of the 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 sensor and actuation delays within *AutoControl* are small compared to the control interval. We also require that the underlying system-level resource schedulers provide rich enough interfaces to dynamically adjust resource shares for the VMs.

3. DESIGN AND IMPLEMENTATION

This section details the design of both AppController and NodeController in the two-layered architecture of *AutoControl*. For easy reference, Table 1 summarizes the mathematical symbols that will be used for key parameters and variables in these controllers.

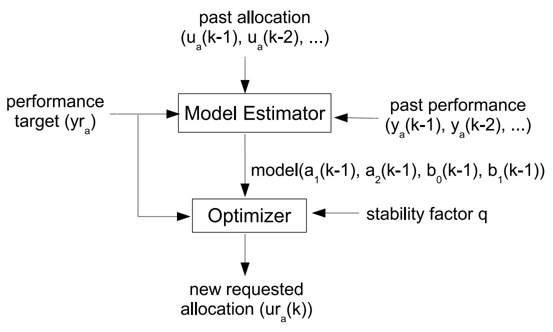

past allocation
(u_a(k-1), u_a(k-2), ...)
performance target (yr_a)
Model Estimator
past performance
(y_a(k-1), y_a(k-2), ...)
model(a_1(k-1), a_2(k-1), b_0(k-1), b_1(k-1))
Optimizer
stability factor q
new requested allocation (ur_a(k))

Figure 4: AppController's internal structure

3.1 Design of AppController

As introduced in Section 2, every hosted application has an AppController associated with it. In order for each AppController to decide how much resource is needed for the application to meet its performance target, it first needs to determine the quantitative and dynamic relationship between the application's resource allocation and its performance. Such a relationship is captured in the notion of "transfer function" in traditional control theory for modeling of 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 can be approximately characterized *locally* by a linear model. We periodically re-estimate the model online based on real-time measurements of the relevant variables and metrics, allowing the model to adapt to different operating regimes and workload conditions.

Every AppController has two modules as illustrated in Figure 4: (1) a *model estimator* that automatically learns in real time a model for the relationship between an application's resource allocation and its performance, and (2) an *optimizer* that predicts the resource allocation required for the application to meet its performance target based on the estimated model. For each application $a \in A$, let $y_a(k)$ be the value of its performance metric provided by an application performance sensor at the end of control interval k, and let yr_a be the desired value for its performance. Furthermore, we define $yn_a(k) = y_a(k)/yr_a$ to be the normalized performance value for interval k. We then define the resource-allocation variable $\mathbf{u}_a$ to be a vector that 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 (we represent all vectors in boldface).

3.1.1 Model estimator

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

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

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. We chose a linear model because it is easy to estimate online and simplifies the corresponding controller design problem. 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. (Some evidence of this will be presented later in Section 5.)

The reason why we model the normalized performance rather than the absolute performance is that the latter can have an arbitrary magnitude. The normalized performance yn_a has values that are comparable to those of the resource allocations in u_a, which are less than 1. This improves the numerical stability of the algorithm.

Note that the model represented in Eq. (1) is *adaptive* itself, because all the model parameters a_1, a_2, $\mathbf{b_0}$ and $\mathbf{b_1}$ are functions of time interval k. These parameters can be re-estimated online using the recursive least squares (RLS) method [4]. At the end of every control interval $k-1$, the model estimator collects the newly-measured performance value $y_a(k-1)$, normalizes it by the performance target yr_a, and uses it to update the values for the model parameters. The approach assumes that drastic variations in workloads that cause significant model parameter changes occur infrequently relative to the control interval, thus allowing the the model estimator to converge locally around an operating point and track changes in the operating point. The recursive nature of RLS makes the time taken for this computation negligible for control intervals longer than 10 seconds.

3.1.2 Optimizer

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

$$J_a = (yn_a(k) - 1)^2 + q\|\mathbf{ur_a}(k) - \mathbf{u_a}(k-1)\|^2. \quad (2)$$

To explain the intuition behind this function, we define $J_p = (yn_a(k) - 1)^2$, and $J_c = \|\mathbf{ur_a}(k) - \mathbf{u_a}(k-1)\|^2$. It is easy to see that J_p is 0 when $y_a(k) = yr_a$, 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 allocation, as described by Eq. (1), we can derive the resource allocation required to minimize the cost function J_a, in terms of the recent resource allocation u_a and the corresponding normalized performance values yn_a:

$$\begin{aligned}\mathbf{ur_a^*}(k) = (\mathbf{b_0}\mathbf{b_0}^T + qI)^{-1}((1 - a_1\ yn_a(k-1)\\ -a_2\ yn_a(k-2) - \mathbf{b_1}^T\mathbf{u_a}(k-1))\mathbf{b_0} + q\mathbf{u_a}(k-1))\end{aligned} \quad (3)$$

This is a special case of the optimal control law derived in [18]. 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 $\Delta yn_a(k) = 1 - a_1\ yn_a(k-1) - a_2\ yn_a(k-2) - \mathbf{b_1}^T\mathbf{u_a}(k-1)$. This indicates the discrepancy between the model-predicted value for $yn_a(k)$ and its target (which is 1) that needs to be compensated by the next allocation ($\mathbf{u_a}(k)$). For a small q value, $\mathbf{ur_a^*}(k)$ is dominated by the effect of $\Delta yn_a(k)$, and the controller reacts agressively to tracking errors in performance. As the value of q increases, $\mathbf{ur_a^*}(k)$ is increasingly dominated by the previous allocation ($\mathbf{u_a}(k-1)$), and the controller responds slowly to the tracking error with less oscillation in the resulting resource allocation. In the extreme of an infinitely large q value, we have $\mathbf{ur_a^*}(k) = \mathbf{u_a}(k-1)$, meaning the allocation remains 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, 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 resources where the total resources requested are less than the available resources, the NodeController divides each resource in proportion to the requests from the AppControllers. For resources that are contested, that is, where the sum of the resource requests is greater than the available resource, the NodeController picks an allocation that locally minimizes the discrepancy 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 application performance, where each application's weight represents its priority relative to other applications.

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

$$ur_{1,cpu,web} + ur_{2,cpu} \leq 1 \quad (4)$$

$$ur_{1,disk,web} + ur_{2,disk} \leq 1 \quad (5)$$

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 four possible scenarios for the virtualized node n1, and the NodeController algorithm for dealing with each scenario.

3.2.1 Scenario I: No CPU or disk contention

In this case, the node has adequate CPU and disk resources to meet all resource requests, and hence the resources are divided in proportion to the resource requests, as follows:

$$
\begin{aligned}
u_{1,cpu,web} &= ur_{1,cpu,web}/(ur_{1,cpu,web}+ur_{2,cpu}) && (6)\\
u_{2,cpu} &= ur_{2,cpu}/(ur_{1,cpu,web}+ur_{2,cpu}) && (7)\\
u_{1,disk,web} &= ur_{1,disk,web}/(ur_{1,disk,web}+ur_{1,disk}) && (8)\\
u_{2,disk} &= ur_{2,disk}/(ur_{1,disk,web}+ur_{2,disk}) && (9)
\end{aligned}
$$

This allocation policy implies two things: (1) for each application and each resource, the requested allocation should be satisfied; (2) the excess capacity for each resource is allocated to both applications in proportion to their requests.

3.2.2 Scenario II: CPU contention only

In this scenario, node n1 has enough disk resource to meet the requests from the AppControllers, but not enough CPU resource; that is, constraint (4) is violated while constraint (5) is satisfied. The NodeController divides the disk resources in proportion to the requests, as in the previous case, using Eqs. (8) and (9). However, the applications will receive less CPU resource than requested; let us denote the deficiencies as $\Delta u_{1,cpu,web} = ur_{1,cpu,web} - u_{1,cpu,web}$ and $\Delta u_{2,cpu} = ur_{2,cpu} - u_{2,cpu}$. The resulting discrepancy between the achieved and target normalized performance of application 1 can then be estimated as $|\partial yn_1/\partial u_{1,cpu,web}\Delta u_{1,cpu,web}|$, and similarly for application 2. The sum of weighted mean squared discrepancies of the normalized performance values across the applications can then be estimated as:

$$
\begin{aligned}
J_{n1,cpu} &= w_1(\tfrac{\partial yn_1}{\partial u_{1,cpu,web}}\Delta u_{1,cpu,web})^2\\
&\quad +w_2(\tfrac{\partial yn_2}{\partial u_{2,cpu}}\Delta u_{2,cpu})^2
\end{aligned}
$$

The CPU resource allocation is found by optimizing this overall normalized performance discrepancy:

$$\text{Minimize } J_{n1,cpu} \qquad \text{subject to}$$

$$
\begin{aligned}
\Delta u_{1,cpu,web} + \Delta u_{2,cpu} &\geq ur_{1,cpu,web} + ur_{2,cpu} - 1 && (10)\\
\Delta u_{1,cpu,web} &\geq 0 && (11)\\
\Delta u_{2,cpu} &\geq 0 && (12)
\end{aligned}
$$

Note that constraint (10) is simply the capacity constraint (4), applied to actual allocations. Constraints (11) and (12) ensure that no application is throttled to increase the performance of another application beyond its target. In the minimization objective $J_{n1,cpu}$, 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 $\frac{\partial yn_1}{\partial u_{1,cpu,web}} = b_{0,1,cpu,web}$, and $\frac{\partial yn_2}{\partial u_{2,cpu}} = b_{0,2,cpu}$. Both coefficients can be obtained from the model estimators in the AppControllers for both 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-theshelf quadratic programming solver to compute the solution.

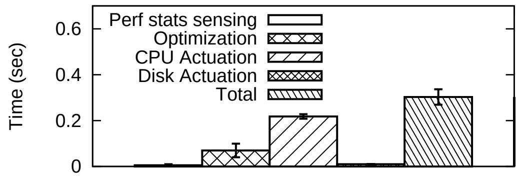


Figure 5: Average performance overhead

3.2.3 Scenario III: Disk contention only

In this scenario, the disk resource is under contention but CPU is not; that is, constraint (5) is violated while constraint (4) is satisfied. The NodeController follows the same policy for CPU allocation as in Scenario I, and solves the following optimization problem to compute the actual disk allocations:

$$
\begin{aligned}
\text{Minimize } J_{n1,disk} &= w_1(\frac{\partial yn_1}{\partial u_{1,disk,web}}\Delta u_{1,disk,web})^2\\
&\quad +w_2(\frac{\partial yn_2}{\partial u_{2,disk}}\Delta u_{2,disk})^2 \qquad \text{s.t.}
\end{aligned}
$$

$$
\begin{aligned}
\Delta u_{1,disk,web} + \Delta u_{2,disk} &\geq ur_{1,disk,web} + ur_{2,disk} - 1 && (13)\\
\Delta u_{1,disk,web} &\geq 0 && (14)\\
\Delta u_{2,disk} &\geq 0 && (15)
\end{aligned}
$$

3.2.4 Scenario IV: CPU and disk contention

This is the scenario where both CPU and disk are under contention. In this scenario, the actual allocations of CPU and disk for both applications will be below the respective requested amounts. The NodeController determines the actual allocations by solving the following optimization problem.

$$
\begin{aligned}
\text{Minimize} \quad J_{n1} &= w_1(\textstyle\sum_{r\in R}\frac{\partial yn_1}{\partial u_{1,r,web}}\Delta u_{1,r,web})^2\\
&\quad +w_2(\textstyle\sum_{r\in R}\frac{\partial yn_2}{\partial u_{2,r}}\Delta u_{2,r})^2.\\
\text{subject to} &\quad \text{Eqs. } (10),(11),(12),(13),(14),(15).
\end{aligned}
$$

Note that the cost function here takes into account the performance degradation of both applications as a result of resource defficiencies in both CPU and disk, and the capacity constraints for both resources need to be considered. 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 show later, empirically, that the performance overhead due to the optimization is small.

3.3 Implementation and performance considerations

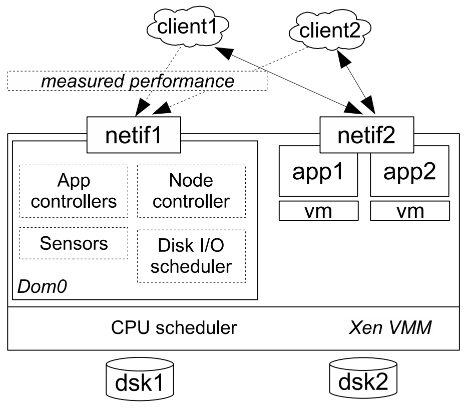


Figure 6: A virtualized node in the testbed

We have implemented the model estimator and the controllers in Java, and written Python wrappers for sensors and actuators provided by the system. The controllers communicate with the clients and the Python wrappers using XML-RPC. The optimization code is written in Matlab and the Java program communicates with Matlab using pipes. A more efficient implementation can be developed using JNI (Java Native call Interface). The total number of lines of code in *AutoControl* is about 3000. Our code is written to be extensible and new controllers can be plugged into the framework easily.

The controllers are designed to be scalable by limiting the number of control variables each controller has to deal with. More specifically, the number of variables for each AppController is the number of tiers multiplied by the number of controlled resources, and the number of variables for each NodeController is the number of VMs on that node multiplied by the number of controlled resources. The performance of each decision in *AutoControl* is mainly affected by three factors: (1) time taken to collect statistics from clients, (2) Matlab optimization time, (3) actuation time. Figure 5 shows the average time taken on our testbed for each of these factors. The total time is less than 1.5% of the control interval.

4. TESTBED AND EXPERIMENTATION

To evaluate *AutoControl*, we have built a testbed consisting of three virtualized nodes, each running multiple VMs hosting multiple applications. Clients running on other nodes generate workloads for these applications.

All the experiments were conducted on 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 hard disks. The blades were installed with OpenSuse 10.3 and we used the default Xen (2.6.22.5-31-xen SMP) available in OpenSuse to run the VMs. The VM images were built using the same distribution, and no changes were made to the kernel.

One network interface and one disk were dedicated to Dom0, which ran the monitoring framework and our controllers. The VMs were allocated the second network interface and disk. The clients connected to the VMs using the network interface dedicated to the VMs. The controller used its own network interface to poll application performance statistics from the clients. In order to demonstrate CPU bottlenecks more easily, we allocated one CPU to the VMs, and used the remaining CPUs for Dom0. Our controller is fully extensible to VMs sharing multiple processors as long as the CPU scheduler allows arbitrary slicing of CPU allocation. Figure 6 shows all the components in our experiments.

These experiments were specifically designed to test the following capabilities of *AutoControl*:

1. Automatically detect and mitigate resource bottlenecks across time and across application tiers;
2. Enforce performance targets for metrics including throughput and average response time;
3. Adapt resource allocations under time-varying application workloads;
4. Prioritize among applications during resource contention.

We used three different applications in our experiments: RUBiS [2], an online auction site benchmark, a Java implementation of the TPC-W benchmark [6], 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 and time-varying intensity. For RUBiS, we used a workload mix called the browsing mix that simulates a user browsing through an auction site. For TPC-W, we used the shopping mix, which simulates a user browsing through a shopping site. The browsing mix stresses the web tier, while the shopping mix exerts more demand on the database tier.

The custom-built 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. A closed-loop client model is used where a new file is only requested after the previous request is complete. Reading a file from the disk consumes disk I/O resource, and encryption requires CPU resource. For a given number of threads,

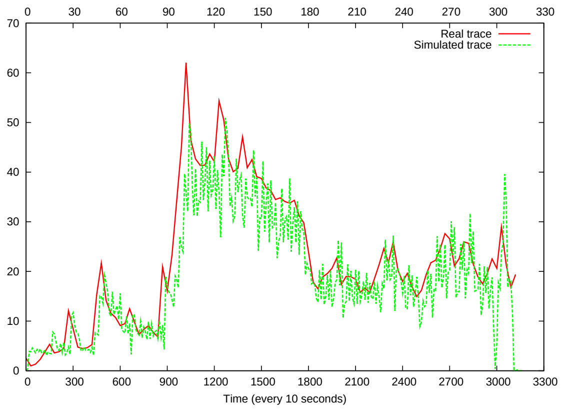


Figure 7: Simulating production traces

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.

4.1 Simulating production traces

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 production at a customer data center. These traces were collected every 5 minutes using the *HP OpenView MeasureWare* agents. We dynamically varied the number of concurrent threads for RUBiS, TPC-W or smedia to recreate the 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 for RUBiS. Note that we only matched the CPU utilization in the production trace. We did not attempt to recreate the disk utilization, because the traces did not contain the needed metadata.

Figure 7 shows the result of simulating the SAP application server CPU utilization using RUBiS. You can see that the CPU utilization in the production trace is closely followed by carefully selecting the number of threads in RUBiS that produce similar CPU consumption.

We also used traces generated from a media workload generator called MediSyn [25]. 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.

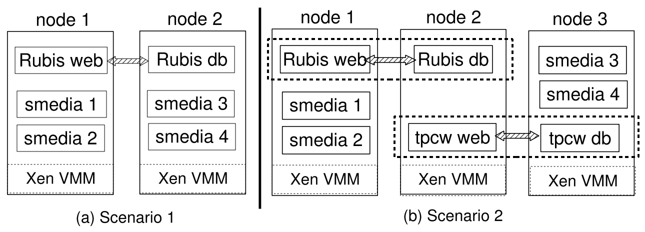


Figure 8: Experimental setup

4.2 Sensors

Our sensors periodically collect two types of statistics: real-time resource utilizations and performance of applications. CPU utilization statistics are collected using Xen's xm command. Disk utilization statistics are collected using the `iostat` command, which is part of the `sysstat` package. In our testbed, we measured both the application throughput and the server-side response time directly from the application, where throughput is defined as the total number of client requests serviced, and for each client request, response time is defined as the amount of time taken to service the request. In a real data center, application-level performance may be obtained from application logs or using tools like *HP OpenView*.

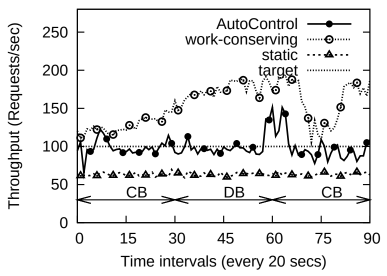
4.3 Actuators

Our actuators included Xen's credit-based CPU scheduler and a custom-built proportional disk I/O scheduler. The credit scheduler provided by Xen allows each domain (or VM) to be assigned a *cap*. We used the cap to specify a CPU share for each VM. This non-work-conserving mode of CPU scheduling provided better performance isolation among applications running in different VMs. The proportional share scheduler for disk I/O was designed to maximize the efficiency of the disk access [11]. The scheduler is logically interposed between the virtual machines and the physical disks: we implemented it as a driver in the Dom0. The controller interacts with the disk I/O scheduler by assigning a *share* to each VM in every control interval.

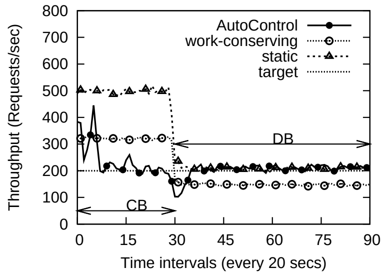
5. EVALUATION RESULTS

We evaluated *AutoControl* in a number of experimental scenarios to answer the questions posed in Section 4. In all of the experiments, a control interval of 20 seconds was used. This control interval was carefully chosen by considering the tradeoff between smaller noise in the sensor measurements (requiring a longer sampling interval) and faster response in the controller (requiring a shorter control interval).

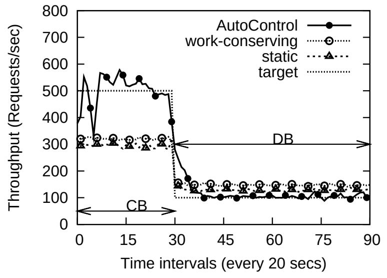
In this section, we present the performance evaluation results from these experiments.



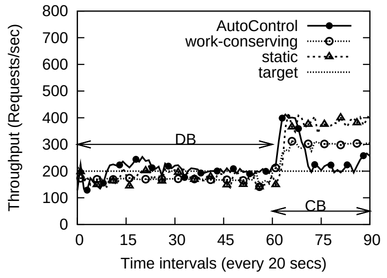
(a) RUBiS throughput



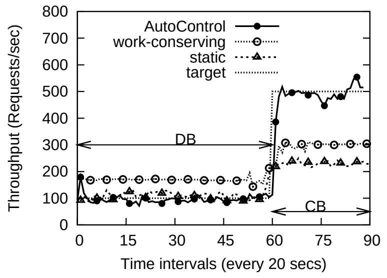
(b) Smedia1 throughput



(c) Smedia2 throughput

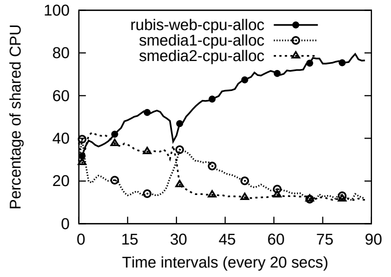


(d) Smedia3 throughput

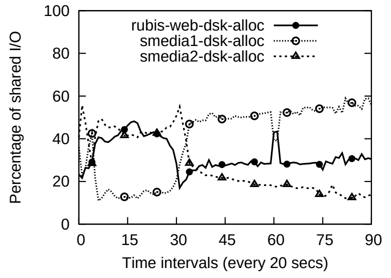


(e) Smedia4 throughput

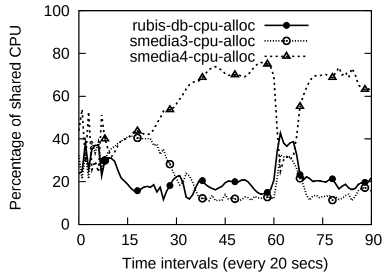
Figure 9: Application throughput with bottlenecks in CPU or disk I/O and across multiple nodes. The time periods with a CPU bottleneck are labeled as "CB" and those with a disk bottleneck are labeled as "DB."



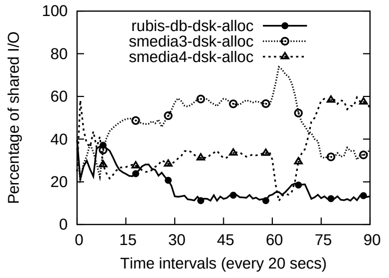
(a) CPU allocations on node 1



(b) Disk allocations on node 1



(c) CPU allocations on node 2



(d) Disk allocations on node 2

Figure 10: Resource allocations to different applications or application tiers on different nodes.

5.1 Scenario 1: Detecting and mitigating resource bottlenecks in multiple resources and across multiple application tiers

This scenario was designed to validate the following claims about *AutoControl*:

- **Claim 1:** It can automatically detect resource bottlenecks and allocate the proper amount of resources to each application such that all the applications can meet their performance targets if possible. This occurs for different types of resource bottlenecks that occur over time and

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%

across multiple tiers of an application.

- **Claim 2:** It can automatically detect the shift of a bottleneck from one type of resource to another, and still allocate resources appropriately to achieve application-level goals.

We use the experimental setup shown in Figure 8(a), where two physical nodes host one RUBiS application spanning two nodes, and four smedia applications. For RUBiS, we used the default browsing mix workload with 600 threads emulating 600 concurrent clients connecting to the RUBiS server, and used 100 requests/sec as the throughput target. Each of the smedia applications was driven with 40 threads emulating 40 concurrent clients downloading media streams at 350KB/sec. We ran calibration experiments to measure the total throughput achievable for each smedia application alone. We observe that, with 50% of clients requesting encrypted streams, the application is CPU-bound and the maximum throughput is just above 700 requests/sec. If, however, only 2% of clients are requesting encrypted streams, the application becomes disk-bound and the maximum throughput is around 300 requests/sec.

We then ran an experiment for 90 control intervals and varied the percentage of encrypted streams to create a shift of the resource bottleneck in each of the virtualized nodes. Table 2 illustrates these transitions. For the first 29 intervals, smedia1 and smedia2 on node 1 were CPU-bound, whereas smedia3 and smedia4 on node 2 were disk-bound. We considered a scenario where smedia1 and smedia3 always had a throughput target of 200 requests/sec each. We then set the throughput targets for smedia2 and smedia4 at 500 and 100 requests/sec, respectively. At interval 30, smedia1 and smedia2 on node 1 were switched to disk-bound, and so the throughput target for smedia2 was changed to 100 requests/sec. At interval 60, smedia3 and smedia4 on node 2 were switched to CPU-bound, and so the throughput target for smedia4 was adjusted to 500 requests/sec. The targets were chosen such that both nodes were running near their capacity limits for either CPU or disk I/O.

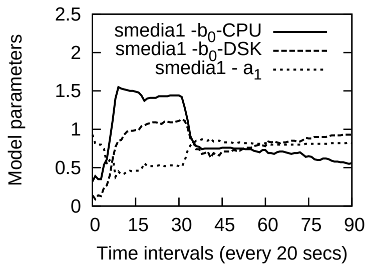
Figure 9 shows the throughput of all the five applications as functions of the control interval. For the first 29 intervals, the RUBiS web tier, smedia1 and smedia2 contended for CPU on node 1, and the RUBiS db tier, smedia3 and smedia4 contended for disk I/O on node 2. *AutoControl* was able to achieve the targets for all the applications in spite of the fact that (i) the resource bottleneck occurs either in the CPU or in the disk; (ii) both tiers of the RUBiS application distributed across two physical nodes experienced resource contention.

To help understand how the targets were achieved, Figures 10(a) and 10(b) show the CPU and disk I/O allocations to the RUBiS web tier, smedia1 and smedia2 on node 1. For the first 29 intervals, these three VMs were contending for CPU. The controller gave different portions of both CPU and disk resources to the three VMs such that all of their targets could be met. In the same time period (first 29 intervals), on node 2, the RUBiS database tier, smedia3 and smedia4 were contending for the disk I/O. Figures 10(c) and 10(d) show the CPU and disk I/O allocations for all 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.

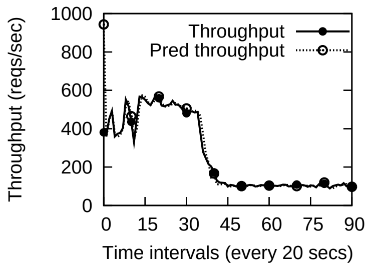
At interval 30, the workloads for the smedia applications on node 1 were switched to be disk-heavy. As a result, smedia1 and smedia2 were contending for disk I/O, since RUBiS web tier uses minimal disk resource. The controller recognized this change in resource bottleneck automatically and ensured that both smedia1 and smedia2 could meet their new throughput targets by allocating the right amount of disk resources to both smedia applications (see Figure 10(b)).

At interval 60, the workloads for the smedia applications on node 2 were switched to be CPU-heavy. Because the RUBiS db tier also requires a non-negligible amount of CPU (around 20%), smedia3 and smedia4 started contending for CPU with the RUBiS db tier on node 2. Again, the controller was able to automatically translate the application-level goals into appropriate resource allocations to the three VMs (see Figure 10(c)).

For comparison, we repeated the same experiment using two other resource allocation methods that are commonly used on consolidated infrastructure, a work-conserving mode and a static 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 20:50:30. The resulting application performance from both approaches is shown in Figure 9 along with the performance from *AutoControl*. As can be seen, neither approach was able to offer the degree of perfor-



(a) Model parameter values for smedia1



(b) Measured and model-predicted throughput for smedia2

Figure 11: Internal workings of the *AppController* - model estimator performance

mance 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 throughput 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 both CPU and disk I/O that will guarantee the performance targets for all the applications. We chose to allocate both CPU and disk resources in the ratio 20:50:30, as a human operator might.

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 allocation. Our online estimator continuously adapts the model parameters as dynamic changes occur in the system. Figure 11(a)(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,

Table 3: Predictive accuracy 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

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.

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

$$R^2 = 1 - \frac{\sum_k (\hat{yn}_a(k) - yn_a(k))^2}{\sum_k (yn_a(k) - yn_{a,avg})^2}$$

$$MAPE = \frac{1}{K} \sum_{k=1}^{K} \left| \frac{\hat{yn}_a(k) - yn_a(k)}{yn_a(k)} \right|$$

where K is the total number of samples, $\hat{yn}_a(k)$ and $yn_a(k)$ are the model-predicted value and the measured value for the normalized performance of application a, and $yn_{a,avg}$ is the sample mean of yn_a. Table 3 shows the values of these two measures for all the five applications. As an example, we also show in Figure 11(b)(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 Scenario 2: Enforcing application priorities

In this scenario, we use the experimental setup shown in Figure 8(b) to substantiate the following two claims:

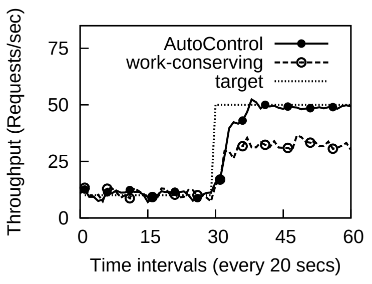
- **Claim 3:** *AutoControl* can support different multi-tier applications.
- **Claim 4:** During resource contention, if two applications sharing the same resource have different priority weights, the application with a higher priority weight will see a lower normalized tracking error ($|yn_a - 1|$) in its performance.

In this setup, we have two multi-tier applications, RUBiS and TPC-W, and four smedia applications spanning three nodes. We ran the same workloads used in Scenario 1 for RUBiS, smedia1 and smedia2. TPC-W was driven with the shopping mix workload with 200 concurrent threads. Each of the smedia applications on node 3 was driven with a workload of 40 concurrent users, where 50% of clients requested encrypted streams (making it CPU-bound). We assume that the TPC-W application is of higher priority than the two smedia applications sharing the same node. Therefore, TPC-W is assigned a priority weight of $w = 2$ while the other applications have $w = 1$ in order to provide service differentiation.

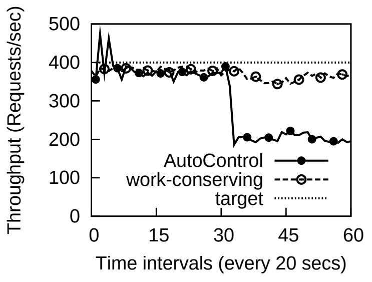
Unlike the setup used in Scenario 1, there was no resource contention on node 2. For the first 29 intervals, all the six applications were able to meet their goals. Figure 12 shows the throughput target and the achieved throughput for TPC-W and smedia3. (The other four applications are not shown to save space.) At interval 30, 800 more threads were added to the TPC-W client, simulating increased user activity. The throughput target for TPC-W was adjusted from 20 to 50 requests/sec to reflect this change. This increases the CPU load on the database tier creating a CPU bottleneck on node 3. *AutoControl* responds to this change automatically and correctly re-distributes the resources. Note that not all three applications (TPC-W, smedia3, and smedia4) on node 3 can reach their targets. But 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 12. In this mode, smedia3 and smedia4 took up more CPU resource, causing TPC-W 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 13 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 12. 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.

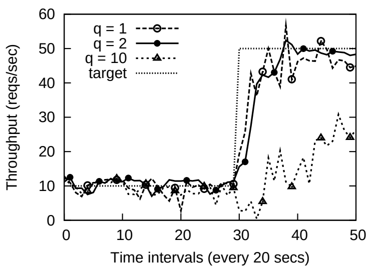


(a) TPC-W throughput

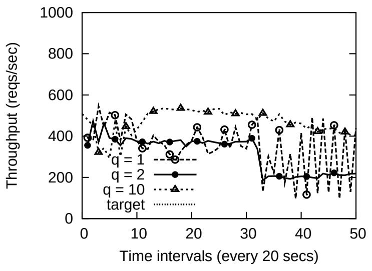


(b) Smedia3 throughput

Figure 12: 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 13: Performance results for TPC-W and smedia3 with stability factor $q = 1, 2, 10$

5.3 Scenario 3: Production-trace-driven workloads

The last scenario is designed to substantiate the following two claims for *AutoControl*:

- **Claim 5:** It can be used to control application response time.
- **Claim 6:** It can manage workloads that are driven by real production traces.

In this scenario, we use the same setup as in Scenario 1, shown in Figure 8(a). We simulated the production workloads using the trace-driven approach as described in Section 4. Each of the RUBiS, smedia1 and smedia2 applications was driven by a production trace, while both smedia3 and smedia4 run a workload with 40 threads with a 2% chance of requesting

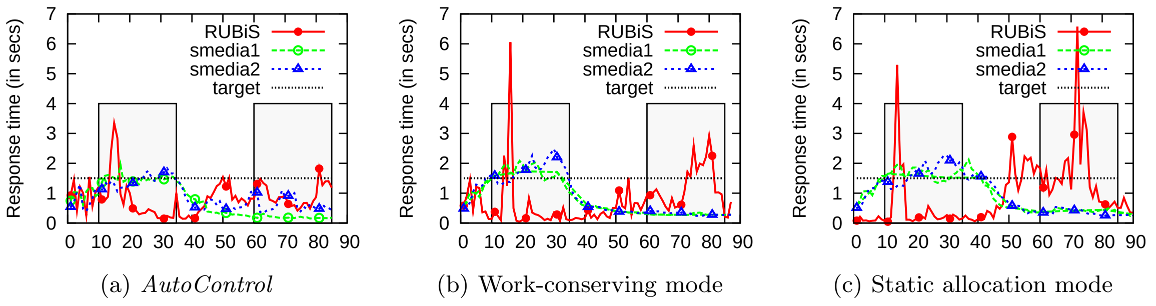


Figure 14: Performance comparison of *AutoControl*, work-conserving mode and static allocation mode, while running RUBiS, smedia1 and smedia2 with production-trace-driven workloads.

an encrypted stream (making it disk-bound).

In this experiment, we use response time as the performance metric for two reasons.

1. Response time behaves quite nonlinearly with respect to the resource allocation and can be used to evaluate how *AutoControl* copes with nonlinearity in the system.

2. It is possible to specify the same response time target for all applications even if they are different, whereas specifying throughput targets are harder since they may depend on the offered load.

For brevity, we only show the results for the three applications running on node 1. Figures 14(a), 14(b) and 14(c) show the measured average response time of RUBiS, smedia1 and smedia2 as functions of the control interval, using *AutoControl*, work-conserving mode, and static allocation mode, respectively. We use 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 to both smedia1 and smedia2 higher shares of the CPU 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 any of the three applications.

The performance 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 allocation, and that the real traces used here were much more dynamic than the static workloads with step changes tested in Scenario 1 and 2, *AutoControl* was still able to balance the resources and minimize the response time violations for all three applications.

6. RELATED WORK

In recent years, control theory has been applied to computer systems for resource management and performance control [13, 16]. Examples of its application include web server performance guarantees [1], dynamic adjustment of the cache size for multiple request classes [19], CPU and memory utilization control in web servers [9], adjustment of resource demands of virtual machines based on resource availability [28], and dynamic CPU allocations for multitier applications [18, 22]. These concerned themselves with controlling only a single resource (usually CPU), used mostly single-input single-output (SISO) controllers (except in [9]), 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 [9], 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 [7, 10, 14, 20]. 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 [17] to achieve performance differentiation, and efficient adaptive proportional share scheduler [11] for storage systems.

Traditional admission control to prevent computing systems from being overloaded has focused mostly on web servers. Control theory was applied in [15] for the design of a self-tuning admission controller for 3-tier web sites. In [17], 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 the 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.

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 [3] 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 [8]. The active server set of each service was dynamically resized adapting to the offered load. In [24], 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.

7. DISCUSSION AND FUTURE WORK

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

7.1 Migration for dealing with bottlenecks

Migration of VMs can also be used to handle an overloaded physical node [27]. The black box migration strategy uses resource-utilization statistics to infer which VMs need to be migrated, and may not know about SLO violations. The gray box migration strategy uses application statistics to infer the resource requirements using queuing theory. However, requirements for complex applications requiring multiple resources cannot be easily predicted using queuing theory. In addition, migration of stateful applications (e.g., databases) might take too long to mitigate an SLO violation; in fact, the impact on the SLOs is compounded during the migration. In a heavily-consolidated data center where most of the nodes are highly utilized, migration may not be viable. Finally, security concerns in migration [21] may cause the vendors to add security features that will make migration much slower.

Despite these shortcomings, migration is useful when there is a long-term mismatch between node resources and application requirements. Migration can be added as an actuator to our system. The migration actuator works at a much more coarse-grained time-scale, and as future work, we plan to extend *AutoControl* to utilize the migration actuator along with the resource allocation schedulers.

7.2 Actuator & sensor behavior, network and memory control

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 [12] 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.

Our intial efforts in adding network resource control have failed, because of inaccuracies in network actuators. Since Xen's native network control is not fully implemented, we tried to use Linux's existing traffic controller (`tc`) to allocate network resources to VMs. We found that the network bandwidth setting in (`tc`) is not enforced correctly when heavy network workloads are run. We plan to fix these problems and add network control as well. The theory we developed in this paper is directly applicable to any number of resources.

The memory ballooning supported in VMware [26] provides a way of controlling the memory required by a VM. However, the ballooning algorithm does not know about application goals or multiple tiers, and only uses the memory pressure as seen by the operating system. In the future, we also plan to add memory control actuators to our system.

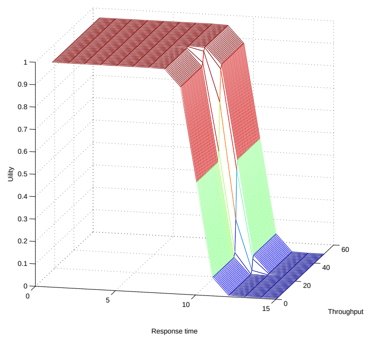


Figure 15: Combined metrics, $thr_{ref} = 25$, $rt_{ref} = 10$, $\alpha = 1$, $\beta = 1$, $thr = 0 - 50$, $rt = 0 - 15$

7.3 Handling a combination of multiple targets

AutoControl is shown to be able to handle different metrics, including throughput and response time as performance metrics. *How do we handle applications that want to specify a combination of metrics for performance?* We have developed a preliminary utility-based framework, where we introduce the concept of utility that is a representative of the "value" of application. For example, a utility function like $U(y) = max\{\alpha(1 - e^{y-y_{ref}}, 0)\}$ represents higher utility for an application as it reaches its target metric y_{ref}. Continuing this, we can create a utility function using two or more metrics that are of interest to the application. An example utility function using both response time and throughput is: $U(thr, rt) = g(rt - rt_{ref} + f(thr_{ref} - thr))$, where $g(x) = (1 + erf(\alpha * x))/2$ and $f(x) = \beta * g(x)$.

A 3-D visualization of the function is shown in Figure 15.

8. CONCLUSIONS

In this paper, we presented *AutoControl*, a feedback control system to dynamically allocate computational resources to applications in shared virtualized environments. *AutoControl* consists of an online model estimator that captures the relationship between application-level performance and resource allocation and a novel MIMO resource controller that determines appropriate allocation of multiple resources to achieve application-level SLOs. We evaluated *AutoControl* using a testbed consisting of Xen virtual machines and various single-tier and multi-tier applications and benchmarks. Our experimental results confirm that *AutoControl* can detect CPU and disk bottlenecks across multiple nodes and can adjust resource allocation to achieve end-to-end application-level SLOs. In addition, *AutoControl* can cope with shifting resource bottlenecks and provide a level of service differentiation according to the priority of individual applications. Finally, we showed that *AutoControl* can enforce performance targets for different application-level metrics, including throughput and response time.

9. REFERENCES

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

[2] Amza, C., Ch, A., Cox, A., Elnikety, S., Gil, R., Rajamani, K., Cecchet, E., and Marguerite, J. Specification and implementation of dynamic Web site benchmarks. In *Proc. of IEEE 5th Annual Workshop on Workload Characterization* (Oct. 2002).

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

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

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

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

[7] Chambliss, D., Alvarez, G., Pandey, P., Jadav, D., Xu, J., Menon, R., and Lee, T. Performance virtualization for large-scale storage systems. In *Proc. of Symp. on Reliable Distributed Systems (SRDS)* (Oct. 2003), pp. 109–118.

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

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

[10] Goyal, P., Modha, D., and Tewari, R. CacheCOW: providing qoS for storage system caches. In *Proc. of ACM SIGMETRICS* (2003), pp. 306–307.

[11] Gulati, A., Merchant, A., Uysal, M., and Varman, P. Efficient and adaptive proportional share I/O scheduling. Tech. Rep. HPL-2007-186, HP Labs, Nov 2007.

[12] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in xen. In *Proc. of International Middleware Conference* (2006), vol. 4290, pp. 342–362.

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

[14] Jin, W., Chase, J., and Kaur, J. Interposed proportional sharing for a storage service utility. In *Proc. of ACM SIGMETRICS* (2004), pp. 37–48.

[15] Kamra, A., Misra, V., and Nahum, E. Yaksha: A self-tuning controller for managing the

performance of 3-tiered web sites. In *Proc. of the International Workshop on Quality of Service (IWQoS)* (June 2004).

[16] Karamanolis, C., Karlsson, M., and Zhu, X. Designing controllable computer systems. In *Proc. of HOTOS* (June 2005), pp. 49–54.

[17] Karlsson, M., Karamanolis, C., and Zhu, X. Triage: Performance isolation and differentiation for storage systems. In *Proc. of IEEE Int. Workshop on Quality of Service (IWQoS)* (2004).

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

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

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

[21] Oberheide, J., Cooke, E., and Jahanian, F. Empirical exploitation of live virtual machine migration. In *Proc. of BlackHat DC convention* (2008).

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

[23] Rolia, J., Cherkasova, L., Arlit, M., and Andrzejak, A. A capacity management service for resource pools. In *Proc. of the 5th International Workshop on Software and Performance (WOSP)* (July 2005).

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

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

[26] Waldspurger, C. Memory resource management in VMware ESX server. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2002).

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

[28] Zhang, Y., Bestavros, A., Guirguis, M., Matta, I., and West, R. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proc. of the Virtual Execution Environments, VEE* (2005), pp. 2–12.