PARTIC: Power-Aware Response Time Control for Virtualized Web Servers*

Yefu Wang, Xiaorui Wang, Ming Chen
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN 37996
{ywang38, xwang, mchen11}@utk.edu

Xiaoyun Zhu
VMware Incorporation
Palo Alto, CA 94304
xzhu@vmware.com

Abstract

Both power and performance are important concerns for enterprise data centers. While various management strategies have been developed to effectively reduce server power consumption by transitioning hardware components to lower-power states, they cannot be directly applied to today's data centers that rely on virtualization technologies. Virtual machines running on the same physical server are correlated, because the state transition of any hardware component will affect the application performance of all the virtual machines. As a result, reducing power solely based on the performance level of one virtual machine may cause another to violate its performance specification. This paper proposes PARTIC, a two-layer control architecture designed based on well-established control theory. The primary control loop adopts a multi-input-multi-output control approach to maintain load balancing among all virtual machines so that they can have approximately the same performance level relative to their allowed peak values. The secondary performance control loop then manipulates CPU frequency for power efficiency based on the uniform performance level achieved by the primary loop. Empirical results demonstrate that PARTIC can effectively reduce server power consumption while achieving required application-level performance for virtualized enterprise servers.

1 Introduction

Recent years have seen a rapid growth of enterprise data centers that host thousands of high-density servers and provide outsourced business-critical IT services. There are two key challenges for effectively operating a modern data center. First, service owners need to be assured by meeting their required Service-Level Objectives (SLOs) such as response time and throughput. Second, power consumption has to be minimized in order to reduce operating costs and avoid failures caused by power capacity overload or system overheating due to increasing high server density.

A well-known approach to reducing power consumption of enterprise servers is to transition the hardware components from high-power states to low-power states whenever possible. Most components in a modern computer server such as processors, main memory and disks have adjustable power states. Components are fully operational, but consume more power in high-power states while having degraded functionality in low-power states. An energy-efficient server design is to have run-time measurement and control of desired application performance by adjusting the power states of certain hardware components. As a result, we can have the desired server performance while reducing power consumption to the maximum degree.

However, this approach cannot be directly applied to today's popular virtualized server environments. In recent years, more and more data centers start to adopt server virtualization strategies for resource sharing to reduce hardware and operating costs. Virtualization technologies such as VMware [2], Xen [3] and Microsoft Virtual Servers [4] can consolidate applications previously running on multiple physical servers onto a single physical server, and so effectively reduce the energy consumption of a data center by shutting down unused servers. As a result, virtual machines on the same physical server are correlated, because the state transition of any hardware component of the physical server will affect the performance of all the virtual machines. As some virtual machines may have light workloads while others may have heavy workloads at the same time, reducing server power consumption based on the performance level of a single virtual machine may cause others to violate their performance specifications. Therefore, the performance of all virtual machines has to be controlled simultaneously for intelligent power saving.

This special characteristic of virtualized enterprise servers calls for more advanced Multi-Input-Multi-Output (MIMO) control solutions. Recently, feedback control theory has been successfully applied to power-efficient performance control for non-virtualized servers [5, 6, 7]. In contrast to traditional adaptive management solutions that rely on heuristics, a key advantage of having a *control-theoretic* foundation is its theoretically guaranteed control accuracy and system stability [8]. We can also have well-established approaches to choosing the right control param-

*This is a significantly extended version of a conference paper [1]. This work was supported, in part, by NSF CSR Grant CNS-0720663, NSF CAREER Award CNS-0845390, and a Power-Aware Computing Award from Microsoft Research in 2008.

eters so that exhaustive iterations of tuning and testing are avoided [9]. This rigorous design methodology is in sharp contrast to heuristic-based adaptive solutions that rely on extensive empirical evaluation and manual tuning. However, while existing control-theoretic solutions have shown significant promise, their Single-Input-Single-Output (SISO) control models are too simplistic and thus cannot be used to effectively manage virtualized enterprise systems.

In this paper, we propose Power-Aware Response Time Control (PARTIC), a novel two-layer control architecture to provide response time guarantees for virtualized web servers. The primary control loop adopts a MIMO control strategy to maintain load balancing among all virtual machines so that they can have roughly the same response time relative to their maximum allowed response times. The secondary loop then controls the relative response times of all the virtual machines to a desired level by adapting CPU frequency for power efficiency. Specifically, the contributions of this paper are four-fold:

- We model the response times of the applications running in virtual machines using system identification, and validate the models with data measured in real experiments.
- We design a two-layer control architecture based on control theory and analyze the control performance.
- We present the system architecture of our control loops and the implementation details of each component.
- We present empirical results to demonstrate that PARTIC can effectively reduce server power consumption while achieving desired response time for virtualized web servers.

The rest of the paper is organized as follows. Section 2 introduces the overall architecture of the control loops. Section 3 and Section 4 present the modeling, design and analysis of the load balancing controller and the response time controller, respectively. Section 5 describes the implementation details of our testbed and each component in the control loops. Section 6 presents the results of our experiments conducted on a physical testbed. Section 7 highlights the distinction of our work by discussing the related work. Section 8 concludes the paper.

2 Control Architecture

In this section, we provide a high-level description of the control system architecture of PARTIC, which features a two-layer controller design. As shown in Figure 1, the primary control loop, *i.e.*, the load balancing control loop, maintains load balancing among all the virtual machines by adjusting the CPU resource (*i.e.*, fraction of CPU time) allocated to them. As a result, the virtual machines can have roughly the same *relative* response time, which is defined as the ratio between the actual response time and the maximum allowed response time for each virtual machine. The secondary control loop, *i.e.*, the response time control loop, then controls the relative average response time to the desired level (*e.g.*, 90%) for all the virtual machines by manipulating CPU frequency. Consequently, power efficiency can be achieved for the physical server while the desired response times can also be guaranteed for applications in all the virtual machines.

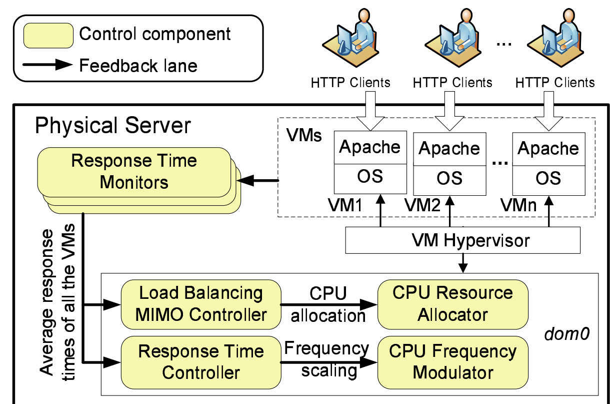


Figure 1. PARTIC's two-layer control architecture.

We choose to control response time instead of other metrics such as resource utilization because response time is a user-perceived metric, which captures users' experience of Internet service performance. In this paper, we control the average response time to reduce the impact of the long delay of any single web request. However, our control architecture can also be applied to control the maximum or 90 percentile response time.

A possible alternative design is to allow different virtual machines to have different CPU frequencies, and throttle the CPU to the desired frequency when the system starts to run each virtual machine. However, doing so without adjusting the CPU time allocation among different virtual machines would reduce the system's adaptation capability, because more CPU time could be allocated to a virtual machine whose response time is longer than the desired set point but its CPU frequency is already at the highest level.

As shown in Figure 1, the key components in the load balancing control loop include a *MIMO controller*, a *response time monitor* for each virtual machine, and a *CPU resource allocator*. The control loop is invoked periodically and its period is selected such that multiple HTTP requests can be received during a control period and the actuation overhead is acceptable. The following steps are invoked at the end of every control period: 1) The response time monitor of each virtual machine measures the average response time in the last control period, and sends the value to the controller; 2) The controller calculates the relative response time for each virtual machine, and the average relative response time of all virtual machines. The difference between the relative response time of each virtual machine and the average value is used as the *controlled variable* with the set point as zero; 3) The controller computes the amount of CPU resource to be allocated to each virtual machine, *i.e.*, the *manipulated variable*, in the next control period; 4) The CPU resource

allocator then enforces the desired CPU resource (*i.e.*, CPU time) allocation. Our control solution can be extended to handle multi-tier web services by modeling the correlations between the virtual machines that run different tiers, which is part of our future work.

The key components in the response time control loop include a *controller*, the same response time monitors and a *CPU frequency modulator*. The control loop is also invoked periodically. In each control period of the response time control loop, the controller receives the average response times of all the virtual machines from the monitors. If we configure the load balancing controller to run on a smaller time scale and enter its steady state by the end of each control period of this controller, all virtual machines should have the same relative response time, which is the *controlled variable* of this loop. Based on this uniform value, the controller then computes the new CPU frequency level, *i.e.*, the *manipulated variable*, for the processor of the physical server, and then sends the level to the CPU frequency modulator. The modulator changes the CPU frequency level of the processor accordingly. When there are too many requests so that the relative average response time is longer than its set point while the CPU frequency is already at the highest level, we conduct admission control to achieve the desired response time by rejecting selected low-priority requests.

Clearly, the two control loops need to be coordinated since they may impact each other. First, the load balancing controller is trying to minimize the difference between the response time of each virtual machine and the average response time of all the virtual machines, while the average response time is being controlled by the response time control loop to the desired set point. Second, every invocation of the response time control loop may change the CPU frequency (thus the total available CPU resource) of the physical server. Therefore, to minimize the influence to the load balancing control loop, the control period of the response time loop is selected to be longer than the settling time of the load balancing control loop. This guarantees that the load balancing loop can always enter its steady state within one control period of the response time control loop, so that the two control loops are decoupled and can be designed independently.

Since the core of each control loop is its controller, we introduce the design and analysis of the two controllers in the next two sections, respectively. The implementation details of other components are given in Section 5.

3 Load Balancing Controller

In this section, we present the problem formulation, modeling, design and analysis of the load balancing controller.

3.1 Problem Formulation

Load balancing control can be formulated as a dynamic optimization problem. We first introduce some notation. A physical server hosts n virtual machines. T_1 is the control period. d_i is the maximum allowed response time of ith virtual machine VM_i. $rt_i(k)$ is the average response time of VM_i in the kth control period. $r_i(k)$ is the relative response time of VM_i. Specifically, $r_i(k) = rt_i(k)/d_i$. $r(k)$ is the average relative response time of all virtual machines in the kth control period, namely, $r(k) = \sum_{i=1}^{n} r_i(k)/n$. $e_i(k)$ is the control error of VM_i, *i.e.*, $e_i(k) = r_i(k) - r(k)$. $e_i(k)$ is our controlled variable and its set point is zero.

To allocate CPU resource to VM_i, the virtual machine hypervisor assigns an integer *weight* $w_i(k)$ to VM_i. The CPU resource is allocated to each virtual machine proportionally to its weight. In order to linearize the system, instead of directly using $w_i(k)$ as the control input, we find a constant operating point w_i for each virtual machine, which is the typical weight of the virtual machine. The typical weights can be chosen such that all the VMs on the server can have approximately the same relative response time under typical workloads. The weight change, namely, $\Delta w_i(k) = w_i(k) - w_i$ is used as our manipulated variable.

Given a control error vector, $\mathbf{e(k)} = [e_1(k) \ldots e_n(k)]^T$, the control goal at the kth control point (*i.e.*, time kT_1) is to dynamically choose a weight change vector $\mathbf{\Delta w(k)} = [\Delta w_1(k) \ldots \Delta w_n(k)]^T$ to minimize $e_i(k+1)$ for all virtual machines:

$$\min_{\{\Delta w_j(k)|1 \le j \le n\}} \sum_{i=1}^{n} (e_i(k+1))^2 \qquad (1)$$

3.2 System Modeling

In order to have an effective controller design, it is important to model the dynamics of the controlled system, namely the relationship between the controlled variables (*i.e.*, $e_i(k), 1 \le i \le n$) and the manipulated variables (*i.e.*, $\Delta w_i(k), 1 \le i \le n$). However, a well-established physical equation is usually unavailable for computer systems. Therefore, we use a standard approach to this problem called *system identification* [8]. Instead of trying to build a physical equation between the manipulated variables and controlled variable, we infer their relationship by collecting data in experiments and establish a statistical model based on the measured data. To establish a system model that accounts for the impacts from the response time control loop, in the system identification, we adopt a typical workload that can be handled by the response time controller. Accordingly, we set the CPU frequency of the server to be 73% of its maximum value, such that the average relative response time can reach the desired set point, as controlled by the response time controller.

Based on control theory [10], we choose to use the following difference equation to model the controlled system:

$$\mathbf{e(k)} = \sum_{i=1}^{m_1} \mathbf{A_i e(k-i)} + \sum_{i=1}^{m_2} \mathbf{B_i \Delta w(k-i)} \qquad (2)$$

where m_1 and m_2 are the orders of the control output and control input, respectively. $\mathbf{A_i}$ and $\mathbf{B_i}$ are control parameters whose values need to be determined by system identification.

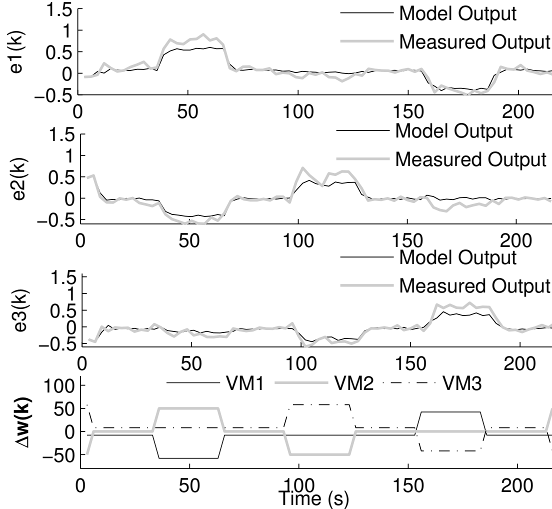


Figure 2. Model validation for the load balancing controller.

Table 1. Model orders and RMSE.

	$m_1 = 0$	$m_1 = 1$	$m_1 = 2$
$m_2 = 1$	0.0930	0.0862	0.0858
$m_2 = 2$	0.0926	0.0860	0.0857
$m_2 = 3$	0.1008	0.0953	0.0949

For system identification, we need to first determine the right orders for the system, *i.e.*, the values of m_1 and m_2 in the difference equation (2). The order values are normally a compromise between model simplicity and modeling accuracy. In this paper, we test different system orders as listed in Table 1. For each combination of m_1 and m_2, we generate a sequence of pseudo-random digital white noise [8] to stimulate the system and then measure the control output (*i.e.*, $e_i(k), 1 \leq i \leq n$) in each control period. Our experiments are conducted on the testbed introduced in detail in Section 5. Based on the collected data, we use the *Least Squares Method (LSM)* to iteratively estimate the values of parameters $\mathbf{A_i}$ and $\mathbf{B_i}$. The values in Table 1 are the estimated accuracy of the models in term of Root Mean Squared Error (RMSE). From Table 1, we can see that the model with $m_1 = 2$ and $m_2 = 1$ has a small error while keeping the orders low. Note that although the model with $m_1 = 2$ and $m_2 = 2$ has a slightly smaller error, it leads to a higher model order and would result in a more complex controller design with a higher computational overhead. We then use a step-like signal to validate the results of system identification. Figure 2 demonstrates that the predicted output of the selected model is sufficiently close to the actual system output. Therefore, the resultant system model from our system identification is:

$$\mathbf{e(k)} = \mathbf{A_1 e(k-1)} + \mathbf{A_2 e(k-2)} + \mathbf{B_1 \Delta w(k-1)} \quad (3)$$

where $\mathbf{A_1}$, $\mathbf{A_2}$ and $\mathbf{B_1}$ are $n \times n$ constant control parameter matrices whose values are determined by system identification. The values used in our experiments are presented in Section 3.4.

3.3 Controller Design

We apply the Linear Quadratic Regulator (LQR) control theory [10] to design the controller based on the system model (3). LQR is an advanced control technique that can deal with coupled MIMO control problems. Compared to other MIMO control techniques such as Model Predictive Control (MPC), LQR has a smaller runtime computational overhead, which is only a quadratic function of the number of VMs on the server. To design the controller, we first convert our system model to a state space model. The state variables are defined as follows:

$$\mathbf{x(k)} = \begin{bmatrix} \mathbf{e(k)} \\ \mathbf{e(k-1)} \end{bmatrix} \quad (4)$$

We then augment the state space model by considering the accumulated control errors for improved control performance:

$$\mathbf{v(k)} = \lambda \mathbf{v(k-1)} + \mathbf{e(k-1)} \quad (5)$$

where λ is the forgetting factor that determines how fast the controller forgets the previous control errors. A smaller λ allows the controller to forget the errors faster. $\lambda = 1$ means that the controller never forgets the errors. In our experiments, we choose $\lambda = 0.8$ as a compromised value. Our final state space model is as follows:

$$\begin{bmatrix} \mathbf{x(k)} \\ \mathbf{v(k)} \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{x(k-1)} \\ \mathbf{v(k-1)} \end{bmatrix} + \mathbf{B \Delta w(k-1)} \quad (6)$$

where

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_1} & \mathbf{A_2} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \lambda \mathbf{I} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B_1} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}.$$

Based on the augmented model (6), we design the LQR controller by choosing gains to minimize the following quadratic cost function:

$$\mathbf{J} = \sum_{k=1}^{\infty} \begin{bmatrix} \mathbf{e^T(k)} & \mathbf{v^T(k)} \end{bmatrix} \mathbf{Q} \begin{bmatrix} \mathbf{e(k)} \\ \mathbf{v(k)} \end{bmatrix} + \sum_{k=1}^{\infty} \mathbf{\Delta w^T(k) R \Delta w(k)} \quad (7)$$

where $\mathbf{Q}$ and $\mathbf{R}$ are weighting matrices that determine the trade-off between the control error and the control cost. This optimization is conducted over an infinite time horizon. The first item in (7) represents the control errors and the accumulated control errors. By minimizing the first item, the closed-loop system can converge to the desired set points. The second item in (7) represents the control penalty. Minimizing the second item ensures that the controller will minimize the changes in the control input, *i.e.*, the weight change of each virtual machine. A general rule to determine the values of $\mathbf{Q}$ and $\mathbf{R}$ is that larger $\mathbf{Q}$ leads to faster response to

workload variations, while larger $\mathbf{R}$ lets the system be less sensitive to system noise. The concrete values of $\mathbf{Q}$ and $\mathbf{R}$ used in our experiments are presented in Section 3.4. The LQR controller can be implemented using the Matlab command `dlqry` to solve the optimization problem (7) and get the controller gain matrix $\mathbf{F}$ as:

$$\mathbf{F} = \begin{bmatrix} \mathbf{K_{P1}} & \mathbf{K_{P2}} & \mathbf{K_I} \end{bmatrix} \quad (8)$$

where $\mathbf{K_{P1}}, \mathbf{K_{P2}}$ and $\mathbf{K_I}$ are constant controller parameters. In the kth control period, given the control error vectors $\mathbf{e(k)}$ and $\mathbf{e(k-1)}$, the control input, *i.e.*, the weight change of each virtual machine can be computed using the following controller equation:

$$\mathbf{\Delta w(k)} = -\mathbf{F}[\mathbf{e(k)} \quad \mathbf{e(k-1)} \quad \mathbf{v(k)}]^T \quad (9)$$

3.4 Controller Implementation

Based on system identification conducted on the testbed introduced in detail in Section 5, our system model is (3) with the following parameters:

$$\mathbf{A_1} = \begin{bmatrix} -0.1595 & 0.0333 & 0.1262 \\ 0.0666 & -0.1970 & 0.1304 \\ 0.0929 & 0.1636 & -0.2566 \end{bmatrix},$$

$$\mathbf{A_2} = \begin{bmatrix} 0.0058 & 0.0401 & -0.0459 \\ 0.0029 & 0.0434 & -0.0462 \\ -0.0087 & -0.0834 & 0.0921 \end{bmatrix},$$

$$\mathbf{B_1} = \begin{bmatrix} -0.0052 & 0.0015 & 0.0010 \\ 0.0031 & -0.0026 & 0.0020 \\ 0.0021 & 0.0011 & -0.0030 \end{bmatrix}.$$

As discussed in Section 3, our LQR controller solves an optimization problem (7). In our experiments, we use $\mathbf{R} = \mathbf{I_3}$ and $\mathbf{Q} = diag(1000, 1000, 1000, 1000, 1000, 1000)$ to give more weight to the control errors and the accumulated control errors than to the control penalty. Our resultant LQR controller is (8) with the following parameter matrices:

$$\mathbf{K_{P1}} = \begin{bmatrix} -18.9231 & 11.0687 & 7.8544 \\ 5.2492 & -12.1427 & 6.8934 \\ 4.3118 & 11.1592 & -15.4710 \end{bmatrix}$$

$$\mathbf{K_{P2}} = \begin{bmatrix} 0.6669 & 1.4513 & -2.1183 \\ 0.0506 & 1.1409 & -1.1915 \\ -0.5073 & -2.6583 & 3.1656 \end{bmatrix}$$

$$\mathbf{K_I} = \begin{bmatrix} -10.8456 & 6.3381 & 4.5075 \\ 3.0353 & -6.4148 & 3.3794 \\ 2.4309 & 5.5782 & -8.0090 \end{bmatrix}$$

The eigenvalues of the closed-loop system are 0.8, $0.6767, 0.7362, 0.1791 \pm 0.3393i$, 0, 0, and $0.1024 \pm 0.0570i$. Since all the eigenvalues are inside the unit circle, the closed-loop system in our experiments is guaranteed to be stable. The stability of a different system to be controlled by our load balancing controller can be analyzed based on the steps presented in the next subsection.

The pseudo-code for load balancing controller is detailed in Algorithm 1.

Algorithm 1 Pseudo-code for load balancing controller

Invoked at the end of each control period
$\mathbf{v(k)}$*: accumulated control error initialized to zero*
$\mathbf{F}$*: load balancing control parameter*
Load-Balancing-Controller $(v(k))$
begin
Collect relative response time $r_i(k), 1 \le i \le n$;
Controller computes $r(k) = \sum_{i=1}^{n} r_i(k)/n$;
Controller computes $e_i(k) = r_i(k) - r(k), 1 \le i \le n$;
Controller computes $\mathbf{\Delta w(k)}$ based on
$\mathbf{\Delta w(k)} = -\mathbf{F}[\mathbf{e(k)} \quad \mathbf{e(k-1)} \quad \mathbf{v(k)}]^T$;
$w_i(k+1) = \Delta w_i(k) + w_i, 1 \le i \le n$;
Truncate new CPU weight $\mathbf{w(k+1)}$ to integers;
CPU frequency modulator assigns CPU allocation;
Update $\mathbf{v(k+1)}$ based on $\mathbf{v(k+1)} = \lambda \mathbf{v(k)} + \mathbf{e(k)}$;
end

3.5 Control Analysis for Model Variation

In this subsection, we analyze the system stability and settling time when the designed MIMO controller is used on a different server with a system model different from the nominal model described by (6). A fundamental benefit of the control-theoretic approach is that it gives us theoretical confidence for system stability, even when the controller is used in a different working condition. We now outline the general process for analyzing the stability of a virtualized server controlled by the load balancing controller.

1. Derive the control inputs $\mathbf{\Delta w(k)}$ that minimize the cost function based on the *nominal* system model described by (6).

2. Conduct automated system identification on the target virtualized server and get an actual system model as:

$$\begin{bmatrix} \mathbf{x(k)} \\ \mathbf{v(k)} \end{bmatrix} = \mathbf{A}' \begin{bmatrix} \mathbf{x(k-1)} \\ \mathbf{v(k-1)} \end{bmatrix} + \mathbf{B}' \mathbf{\Delta w(k-1)} \quad (10)$$

   where $\mathbf{A}'$ and $\mathbf{B}'$ are actual parameter matrices that may be different from $\mathbf{A}$ and $\mathbf{B}$ in (6).

3. Derive the closed-loop system model by substituting the derived control inputs $\mathbf{\Delta w(k)}$ into the actual system model (10). The closed-loop system model is in the form:

$$\begin{bmatrix} \mathbf{x(k)} \\ \mathbf{v(k)} \end{bmatrix} = (\mathbf{A}' - \mathbf{B}'\mathbf{F}) \begin{bmatrix} \mathbf{x(k-1)} \\ \mathbf{v(k-1)} \end{bmatrix} \quad (11)$$

4. Derive the stability condition of the closed-loop system described by (11). According to control theory, the closed-loop system is stable if all the eigenvalues of matrix $(\mathbf{A}' - \mathbf{B}'\mathbf{F})$ locate inside the unit circle in the complex space.

In our stability analysis, we assume the constrained optimization problem is *feasible*, i.e., there exists a set of

CPU resource weights within their acceptable ranges that can make the response time on every virtual machine equal to the average response time. If the problem is infeasible, no controller can guarantee the set points through CPU resource adaptation. A Matlab program is developed to perform the above stability analysis procedure automatically.

Example: We now analyze the stability when the controller is used in an example working condition with a system model different from the nominal model. Compared with the nominal system, the example system has a different workload with 55 concurrent HTTP requests (to emulate 55 clients) and runs at a different CPU frequency level (0.83%). As a result, the example system has different model parameters than those of the nominal model described in Section 3.4 as:

$$\mathbf{A_1}' = \begin{bmatrix} -0.2275 & 0.0296 & 0.1979 \\ -0.0354 & -0.0669 & 0.1023 \\ 0.2629 & 0.0373 & -0.3002 \end{bmatrix},$$

$$\mathbf{A_2}' = \begin{bmatrix} -0.0508 & 0.0049 & 0.0459 \\ -0.0011 & -0.0936 & 0.0947 \\ 0.0519 & 0.0887 & -0.1406 \end{bmatrix},$$

$$\mathbf{B_1}' = \begin{bmatrix} -0.0010 & 0.0011 & -0.0000 \\ 0.0010 & -0.0015 & 0.0005 \\ 0.0001 & 0.0004 & -0.0005 \end{bmatrix}.$$

By following step 1, we derive the control inputs $\mathbf{\Delta w(k)}$ as (9) with the concrete value of $\mathbf{F}$ presented in Section 3.4. In the second step, the automated system identification on the example system gives us the model of the example system with the parameters, as follows:

$$\mathbf{A}' = \begin{bmatrix} \mathbf{A_1}' & \mathbf{A_2}' & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & 0.8\mathbf{I} \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} \mathbf{B_1}' \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}$$

By following steps 3 and 4, we get the eigenvalues of matrix $(\mathbf{A}' - \mathbf{B}'\mathbf{F})$ as: 0.8, 0.79, 0.77, $-0.18 \pm 0.25i$, 0, 0, and $-0.14 \pm 0.37i$. Since they are all inside the unit circle, the example system is stable under the control of our controller.

We now analyze the settling time of a virtualized server controlled by the load balancing controller. Settling time is the time required for the controlled variable (*i.e.*, $e_i(k)$) to reach and then stay within a specified *error band* from the set point (*i.e.*, zero). The analysis of settling time is important because we need to select the control period of the response time controller (introduced in the next section) to be longer than the settling time of the load balancing controller, such that the two controllers can be decoupled, as introduced in Section 2. As the settling time may change when the actual system model changes for different servers, we need to use the worst-case settling time for the control period selection. To analyze the settling time of an example system, we follow steps 1 to 3 of the stability analysis. After we have the closed-loop system model in step 3, we derive the step response of the system by using a step signal as the input to measure the maximum time for all the outputs to reach and then stay within an error band of $\pm 10\%$ around their set points. We then estimate the worst-case settling time by examining various system models with the workload and the CPU frequency changing within their respective typical regions. Based on our analysis, the worst-case settling time of the designed load balancing controller is 12s. Consequently, we choose the control period of the response time controller to be 24s.

4 Response Time Controller

In this section, we present the modeling, design, implementation, and analysis of the response time controller.

4.1 System Modeling

We first introduce some notation. T_2, the control period is selected to be longer than the settling time of the load balancing controller such that all virtual machines can guarantee to have the same response time within T_2. As defined before, $r_i(k)$ is the relative response time of VM_i in the kth control period. $r(k)$ is the average relative response time of all the virtual machines. R_s is the set point, *i.e.*, the desired relative response time for the system. $f(k)$ is the current CPU frequency of the physical server, relative to the highest frequency of the CPU. For example, $f(k) = 1$ means the CPU is currently running at its highest frequency.

In the kth control period, given current average relative response time $r(k)$, the control goal is to dynamically choose a CPU frequency $f(k)$ such that $r(k)$ can converge to the set point R_s after a finite number of control periods.

Similar to the load balancing controller, we adopt system identification to model the system. An accurate model to capture the relationship between $r(k)$ and $f(k)$ is normally nonlinear due to the complexity of computer systems. To simplify the controller design with a linear model, instead of directly using $r(k)$ and $f(k)$ to model the system, we use their respective differences with their operating points, r and f, which are defined as the typical values of $r(k)$ and $f(k)$. Specifically, the control output of the model is $\Delta r(k) = r(k) - r$, the control input is $\Delta f(k) = f(k) - f$, and the set point of the control output is $\Delta R_s = R_s - r$. An example way to choose operating point is to select the middle value of a range of available CPU frequencies as f, and measure the average relative response time with this frequency and use it as r.

Based on system identification, the system model is:

$$\Delta r(k) = a_1 \Delta r(k-1) + b_1 \Delta f(k-1) \qquad (12)$$

where a_i and b_i are model coefficients whose values are presented in Section 4.2. To verify the accuracy of the model, we generate a different sequence of control input, and then compare the actual control output to that predicted by the model. Figure 3 demonstrates that our model is sufficiently close to the actual system with $R^2 = 0.8896$.

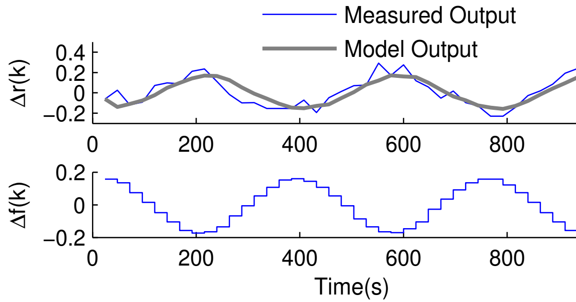


Figure 3. Model validation for the response time controller.

4.2 Controller Design and Implementation

Proportional-Integral (PI) control [8] has been widely adopted in industry control systems. An important advantage of PI control is that it can provide robust control performance despite considerable modeling errors. Following standard control theory, we design our PI controller as:

$$\Delta f(k) = k_1(\Delta R_s - \Delta r(k)) + k_2 \sum_{i=1}^{k}(\Delta R_s - \Delta r(i)) \quad (13)$$

where k_1 and k_2 are control parameters that can be analytically chosen to guarantee system stability and zero steady state error, using standard control design methods [8].

The response time controller is implemented to adjust CPU frequency based on the samples of the response time monitor. Our system model resulted from system identification is (12) with parameters $a_1 = 0.1199$ and $b_1 = -0.8299$. The control period is 24 seconds which is longer than the settling time of the load balancing controller. Based on the system model, our PI controller is designed using the Root-Locus method [8] to guarantee stability and zero steady state error, and also to achieve a settling time of one control period and an overshoot of less than 30%. The final controller is (13) with parameters $k_1 = -0.06$ and $k_2 = -1.7$. The set point of the controller is configured to be 0.88 to give some leeway to the Apache web servers. As a result, the average response time is 88% of the requred soft deadline when the system is in steady state. The poles of the closed-loop system are $-0.17 \pm 0.20i$. As both poles are inside the unit circle, the closed-loop system in our experiments is stable. The stability of a different system to be controlled by the response time controller can be evaluated based on the steps presented in Section 4.3.

4.3 Control Analysis for Model Variation

Although the system has been proven to be stable when the system model (12) is accurate, stability has to be reevaluated when the response time controller is applied to a different server with different workloads. The general process of analyzing the stability of the response time controller with a different system model is similar to that of the load balancing controller.

We first need to derive the control input based on the nominal model (12). System identification can be automatically conducted on the target server to get the actual system model. The closed-loop system model is then built by substituting the derived control inputs into the actual system model. Closed-loop poles can then be calculated based on the closed-loop system model. If all poles are within the unit circle, the target server is stable when it is controlled by the response time controller. The above analysis can be automatically performed by a Matlab program.

Example: We now apply the stability analysis approach to an example system with a heavier workload than the one used to derive our nominal system model. In this system, a workload with 80 concurrent HTTP requests (to emulate 80 clients) is added to every VM. The control input based on the nominal model is shown in Section 5. The system identification on this system gives us its model in the form of (12) with the parameters $a_1 = 0.2402$ and $b_1 = -1.337$. By substituting the derived control inputs into this actual system model, we get the closed-loop poles as -0.17 and -0.94. Since both the poles are inside the unit circle, the example system is stable with our controller.

4.4 Integration with Admission Control

When the system becomes overloaded because of too many requests, the relative average response time can remain longer than its set point even though the CPU frequency is already set to the highest level. As a result, the designed response time controller saturates and thus can no longer achieve the desired response time. In this subsection, we integrate our response time controller with an example admission controller to guarantee the desired response time for high-priority VMs by rejecting selected requests of the low-priority VMs.

In the integrated control solution, all the VMs running on the physical server are categorized into two classes: high-priority VMs and low-priority VMs. In many systems, the priorities can be statically predefined based on the importance of the applications running in the VMs. The priorities can also be dynamically determined based on the current progress of the applications. We use two priority levels just to demonstrate the effectiveness of the integration of response time control and admission control. Our solution can be extended to handle more priority levels. When the system is not overloaded, the response time controller is invoked while the admission controller is disabled. When the system becomes overloaded, the response time controller is disabled because it saturates at the highest relative CPU frequency (*i.e.*, 100%). In the meantime, the example admission controller is enabled to reject a portion of requests received by the low-priority VMs. In the following, we briefly introduce how to switch between the response time controller and the admission controller, and the design of the admission controller.

Controller Switching Condition. The condition of

switching between the two controllers is that the current controller saturates for three consecutive control periods. If the response time controller requires the relative CPU frequency to be higher than 100%, the controller saturates in this control period. If the controller saturates three times consecutively, the system is determined to have an overload. In this case, we set the relative CPU frequency to 100% and switch to the admission controller. On the other side, when the admission controller demands the request rejection rate to be less than 0, the desired response time can be achieved without rejecting any requests. In this case, the rejection rate is set to 0 and the system is switched back to the response time controller for power savings.

Admission Controller Design. In each control period of the admission controller, it measures the current relative average response time $\Delta r(k)$, calculates the difference between $\Delta r(k)$ and the set point ΔR_s, and then computes a new rejection rate $dr(k+1) = \Delta dr(k+1) + dr(k)$ accordingly. The rejection rate $dr(k+1) \in [0, 1]$ is used as the probability for the admission controller to reject each web request received by the low-priority VMs in the next control period. Based on control theory, we design the admission controller as a Proportional (P) controller as shown in (14), where k_{dr} is a constant control parameter. The detailed system modeling and controller design are similar to those of the response time controller described in Section4.2, and skipped due to space limitations.

$$\Delta dr(k+1) = k_{dr}(\Delta R_s - \Delta r(k)) \tag{14}$$

Please note that the focus of this subsection is to demonstrate the integration of our response time control loop with an example admission control scheme. PARTIC can be integrated with more advanced admission control algorithms (*e.g.*, [11]) to handle more sophisticated overload situations. The pseudo-code of the response time controller with the integration of admission control is detailed in Algorithm 2.

Algorithm 2 Pseudo-code for response time controller

Invoke in the end of each control period
Control Mode: either response time control or admission control. Initialized as response time control.
Response-Time-Controller(Control Mode)
begin
  Control Mode = Response Time Control;
  Collect average relative response time $r(k)$;
  if Control Mode == Response Time Control **then**
    Compute desired CPU frequency by (13);
    if Desired CPU frequency ≤ 1 **then**
      Send desired CPU frequency to CPU frequency modulator;
    else
      if Desired CPU frequency > 1 happens for 3 continuous periods **then**
        Change CPU frequency to 1;
        Control Mode = Admission Control;
      end if
    end if
  else
    Compute desired rejection rate by (14);
    if 0<rejection rate ≤ 1 **then**
      Send desired rejection rate to all VMs with LOW priority;
    else
      if Desired rejection rate ≤ 0 **then**
        Change rejection rate to 0;
        Control Mode = Response Time control;
      end if
    end if
  end if
end

5 System Implementation

In this section, we introduce our testbed and the implementation details of the control loops.

5.1 Testbed

Our testbed includes two physical computers. One is named *Server* and hosts three virtual machines (VMs). The other one is named *Client* and emulates a number of HTTP clients that generate HTTP requests to Server. Both computers are running Fedora Core 8 with Linux kernel 2.6.21. Server is equipped with an Intel Xeon X5365 processor and 4GB RAM. The processor has 4 cores and supports 4 frequency levels: 2GHz, 2.33GHz, 2.67GHz and 3GHz. Server and Client are connected via an Ethernet switch.

Xen 3.1 is used as the virtual machine monitor. In Xen, the Xen hypervisor is the lowest layer with the most privileges. When the physical computer and the hypervisor boot, domain 0, *i.e.*, *dom0*, is the first guest operating system that is booted automatically. Dom0 has some special management privileges such as it can direct resource access requests to the hardware. In our testbed, dom0 is used to start the three VMs. In addition, dom0 is used to distribute the HTTP requests from Client to the VMs. The two control loops also run in dom0.

Each VM is allocated 500MB of RAM and 10GB of hard disk space. Each of them is also configured with 4 virtual CPUs, as the physical computer has 4 CPU cores. An Apache server is installed on each VM and runs as a virtualized web server. The Apache servers respond to the HTTP requests from Client with a dynamic web page written in PHP. This example PHP file runs a set of mathematical operations.To be more general, the Apache server on each VM is assigned a different maximum allowed response time, *i.e.*, 180ms for VM1, 190ms for VM2, and 200ms for VM3. In a real system, the maximum allowed response times can be determined by the system administrator based on the requirements of the applications running in the VMs.

The client side workload generator is the Apache HTTP

server benchmarking tool (ab), which is designed to stress test the capability of a particular Apache installation. This tool allows users to manually define the concurrency level, which is the number of requests to perform in a very short time to emulate multiple clients. A concurrency level of 40 is used in our experiments to do system identification and most experiments if not otherwise indicated. Three instances of ab are configured on Client and each one is used to send requests to the Apache Server in a VM.

The power consumption of the physical server is measured with a WattsUp Pro [12] power meter with an accuracy of 1.5% of the measured value. The power meter samples the power data every second and then sends the reading to the controllers through a system file */dev/ttyUSB0*.

5.2 Control Components

We now introduce the implementation details of each component in our control loops.

Response Time Monitor. To eliminate the impact of network delays, we focus on controlling the server-side response time in this paper. The response time monitor is implemented as a small daemon program that runs on each VM. The monitor periodically inserts multiple sample requests into the requests that are sent from Client to the Apache server. Two time stamps are used when a sample request is inserted and when the response is received. The difference is used as the server-side response time, which is sent to the two controllers running in dom0.

CPU Resource Allocator. We use Credit Scheduler [13], Xen's proportional share scheduler to allocate available CPU resource. Credit Scheduler is designed for load balancing of virtual CPUs across physical CPUs on a host. This scheduler allocates CPU in proportion to the integer *weights* assigned to the VMs. In each control period, the load balancing controller computes a *weight* for every VM. The weights are then truncated to integers and given to the VMs.

CPU Frequency Modulator. We use Intel's SpeedStep technology to enforce the desired CPU frequency. In Linux systems, to change CPU frequency, one needs to install the *cpufreq* package and then use root privilege to write the new frequency level into the system file */sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed*. However, this is more complicated with the Xen virtual machine monitor because Xen lies between the Linux kernel and the hardware, and thus prevents the kernel from directly modifying the hardware register based on the level in the system file. As a result, the source code of Xen 3.1 has been hacked to allow the modification of the register.

The Intel CPU used in our experiments supports only four discrete frequency levels. However, the new CPU frequency level periodically received from the response time controller could be any value that is not exactly one of the four supported frequency levels. Therefore, the modulator code must locally resolve the output value of the controller to a series of supported frequency levels to approximate the desired value. For example, to approximate 2.89GHz during a control period, the modulator would output the sequence 2.67, 3, 3, 2.67, 3, 3, etc on a smaller timescale. To do this, we implement a first-order delta-sigma modulator, which is commonly used in analog-to-digital signal conversion. The detailed algorithm of the first-order delta-sigma modulator can be found in [14]. The CPU frequency modulator is a daemon program written in C. It runs in the background and change the CPU frequency according to the out put sequence resulted from the first-order delta-sigma modulator based on the desired CPU frequency. The controller sends a new desired CPU frequency level to the modulator through a named pipe periodically.

Controllers. All the controllers are implemented in dom0 to receive response times from the monitors, and run the control algorithms presented in Section 3 and Section 4, respectively. They are all implemented in a daemon program written in C.

6 Empirical Results

In this section, we first evaluate the performance of the load balancing controller. We then test the two-layer control solution of PARTIC for power efficiency. We then compare PARTIC with a baseline power-efficient solution. Finally, we demonstrate our example admission controller.

We use two baselines for comparison in this paper. OPEN is an open-loop solution that gives fixed CPU resource to each of the three VMs in the system. As a result, OPEN cannot achieve the same relative response time for different VMs when they have non-uniform workloads. OPEN is used in Sections 6.1 and 6.2 to compare with our control solution. SIMPLE uses only the response time controller presented in Section 4 to control the *average* response time of all the VMs, without maintaining load balancing among them. SIMPLE is similar to the control algorithms designed for non-virtualized servers, and is used to demonstrate that those algorithms cannot be directly applied to virtualized computing environments.

6.1 Load Balancing

In this experiment, we disable the response time controller to test the performance of the load balancing controller. As shown in Figure 4(a), the three VMs are initially configured to have the same relative response time. At time 300s, the workload of VM2, increases significantly. This is common in many web applications. For example, breaking news on a major newspaper website may incur a huge number of accesses in a short time. To stress test the performance of our controller in such an important scenario, we double the concurrency level from 40 to 80 for VM2 from time 300s to time 600s to emulate workload increase. Figure 4(a) shows that the load balancing controller can maintain the same relative response time for all the VMs by dynamically increasing the weight of VM2 to allocate more CPU resource to VM2. As the result of load balancing, all the VMs

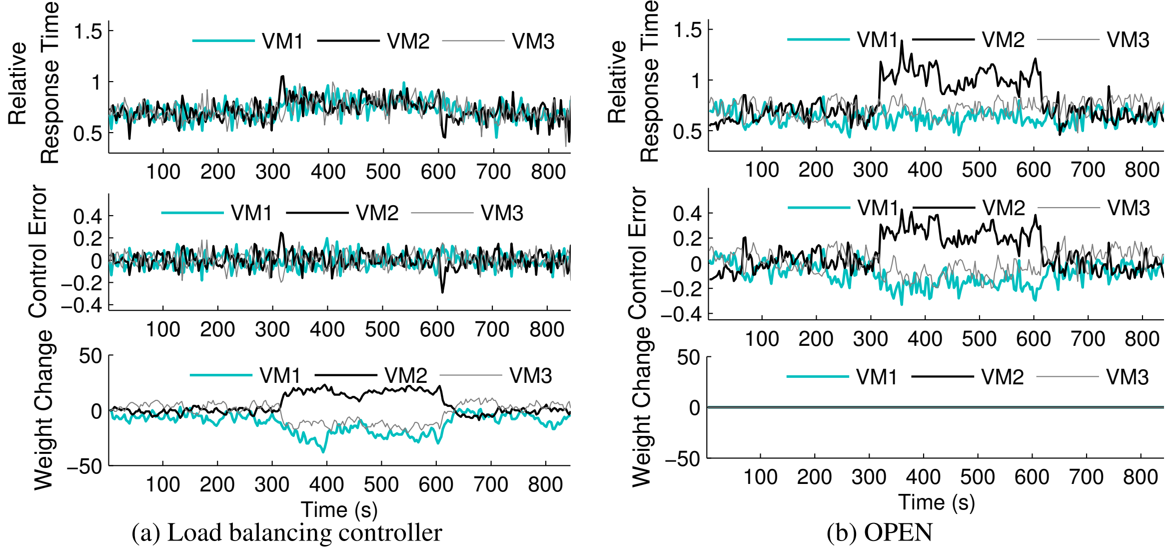


Figure 4. Performance comparison between load balancing controller and OPEN, under a workload increase to VM2 from time 300s to time 600s.

have a slightly longer relative response time but all of them have relative response times lower than 1 after time 300s, which means the desired response times have been guaranteed.

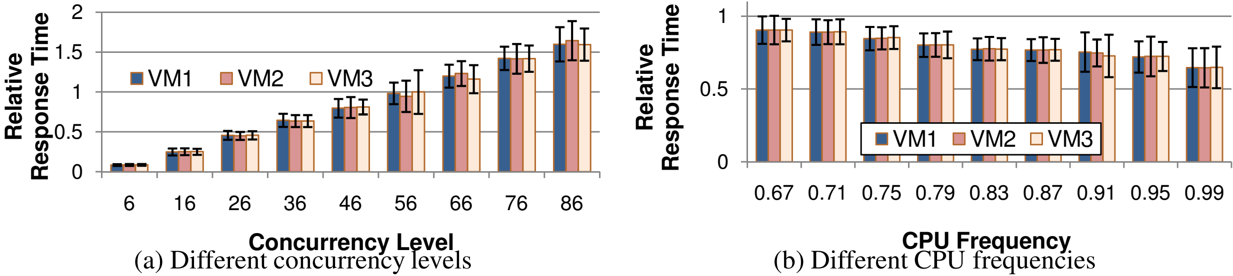
Figure 4(b) shows the performance of OPEN in the same scenario. Clearly, during the workload increase of VM2, the relative response time of VM2 increases significantly while the response times of the other two VMs remain the same. As a result, the relative response time of VM2 is higher than 1 most of the time from time 300s to time 600s, which means that VM2 violates its maximum allowed response time. Under OPEN, the control error between the relative response time of each VM and the average relative response time is also unacceptably large during the workload increase period. The inferior performance of OPEN is caused by its policy of using fixed weights to allocate CPU resource in a static way. Consequently, OPEN cannot adjust CPU resource among different VMs in response to non-uniform workload variations. This experiment demonstrates that the load balancing controller can effectively maintain the same relative response time among all the VMs.

As discussed in Section 3, our system model is the result of system identification with a concurrency level of 40 and a CPU frequency of 0.87. To test the robustness of the load balancing controller when it is used on a system that is different from the one used to do system identification, we conduct two sets of experiments with wide ranges of concurrency level and CPU frequency, respectively. Figure 5(a) shows the means and the standard deviations of the relative response times of the VMs when the concurrency level varies significantly from 6 to 86. Figure 5(b) shows the results when the CPU frequency varies from 0.67 to 0.99. The two figures demonstrate that the load balancing controller works effectively to achieve the same relative response time for all the VMs even when the actual system model is different from the nominal model used to design the controller.

6.2 Power Efficiency

In this experiment, we enable both the load balancing controller and the response time controller to examine power efficiency. We conduct the experiments in the same scenario discussed in the last subsection, with the same workload increase to VM2. Figure 6(a) shows the results of our PARTIC. The experiment starts with the CPU of the physical server running at the highest frequency. As discussed before, the load balancing controller works effectively to achieve the same relative response time for all the VMs. Since the response time is initially unnecessarily shorter than the desired set point, the response time controller lowers the relative CPU frequency from 1 to 0.67 for power efficiency. At time 300s, the workload of VM2 is doubled, which causes the average relative response time to start increasing. The response time controller detects the increase and then adjusts the relative CPU frequency to 0.95, such that the system can run faster to achieve the desired response time. As a result, the average relative response time has been maintained at the desired level. At time 600s when the workload of VM2 is reduced by half, the response time controller throttles the CPU again to achieve power saving while maintaining the desired response time. Throughout the experiment, the average relative response time has been successfully kept to be lower than 1 most of the time, which means the application-level performance, *i.e.*, desired response times have been guaranteed.

Figure 6(b) shows the results of OPEN. OPEN always runs the CPU at the highest frequency, just like most today's computer systems without dynamic power management. From the beginning to time 300s, OPEN has an unnecessarily short average response time at the cost of excessive power consumption. At time 300s when the workload of VM2 is doubled, the average response time increases significantly as OPEN relies on fixed CPU resource allocation



(a) Different concurrency levels

(b) Different CPU frequencies

Figure 5. Performance of the load balancing controller under system model changes.

and CPU frequency. Note that even though it is possible to configure OPEN in a way to achieve power efficiency by running CPU at a lower frequency initially to make the average response time exactly equal to the desired level at the beginning. Doing so may cause the system to violate the maximum allowed response time when the workload increase occurs at time 300s, since OPEN cannot dynamically adjust CPU frequency for desired response time. In this experiment, the average power consumption of the system under OPEN is 233.9 W while it is 205.5 W under PARTIC. On average, with our control solution, a power saving of 28.4 W has been achieved while the desired response time has also been guaranteed.

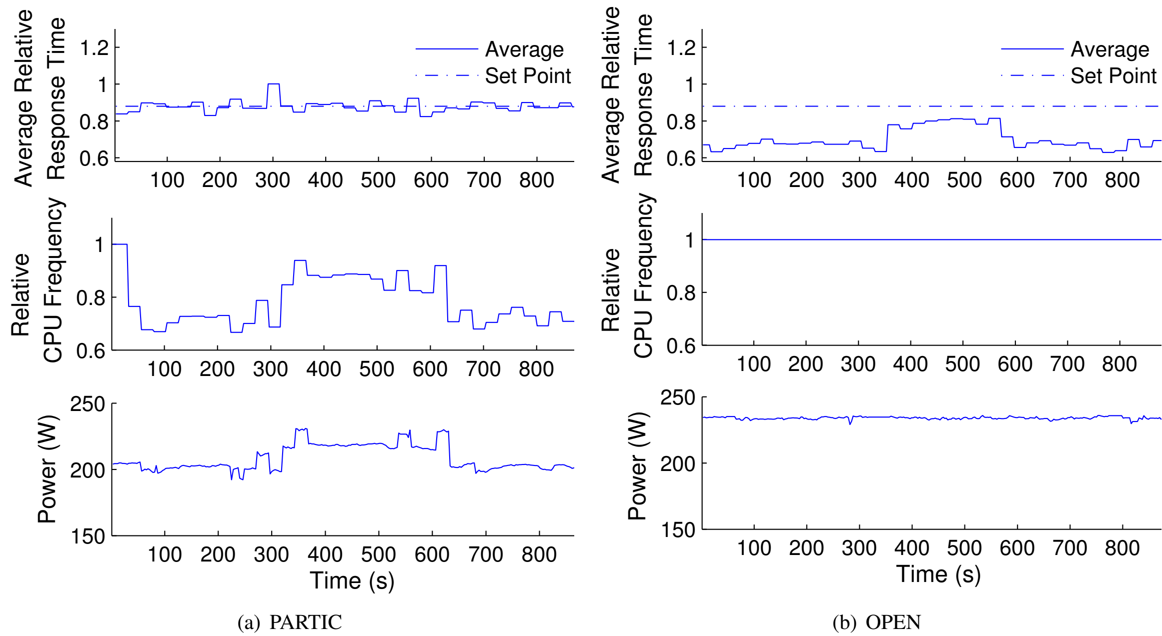
To examine the performance and power efficiency of PARTIC with different workloads, a set of experiments are conducted by changing the concurrency level. As introduced before, higher concurrency level leads to larger number of HTTP requests addressed to each VM. Figure 7(a) shows the performance results. PARTIC can achieve the desired relative response time while OPEN always has unnecessarily short response time, which comes at the cost of excessive power consumption as shown in Figure 7(b). The power consumption of the system under PARTIC is always lower than that under OPEN, and increases as the workload increases to achieve the desired response time. Note that PARTIC has a relative response time shorter than the set point when the concurrency level is 36, as shown in Figure 7(a). That is because the response time controller saturates at the lowest available CPU frequency and thus cannot further increase the response time.

6.3 Comparison to SIMPLE

In this experiment, we compare PARTIC with SIMPLE to show that traditional approaches to power management for non-virtualized servers cannot be easily applied to virtualized computing environments. Figure 8(a) shows the results of PARTIC while Figure 8(b) shows the results of SIMPLE. PARTIC work similarly to control the average relative response time by adjusting CPU frequency except an important difference: PARTIC has a load balancing controller which effectively maintains the same relative response time among all the VMs. For both solutions, the desired average response times have been achieved at the beginning with considerable power saving. However, when VM2 has a doubled workload at time 300s, the response time of VM2 under SIMPLE significantly violates the maximum allowed value. This is because SIMPLE only controls the average relative response time of all the VMs without load balancing. In contrast, PARTIC quickly allocates more CPU resource to VM2 for load balancing after VM2 starts to violate the maximum allowed response time at time 300s. PARTIC then adjusts CPU frequency based on the uniform relative response time achieved by load balancing. As a result, all the VMs achieve their desired response times. Note that SIMPLE has an average power consumption of 203.3W in this particular run, which is slightly lower than that of PARTIC (205.5W) due to noise in the system and power measurement. However, our repeated experiments show that there is no evident statistical difference between the power consumptions of the two schemes. This experiment demonstrates that maintaining load balancing among all VMs is important for power management in virtualized enterprise servers.

6.4 Integration with Admission Control

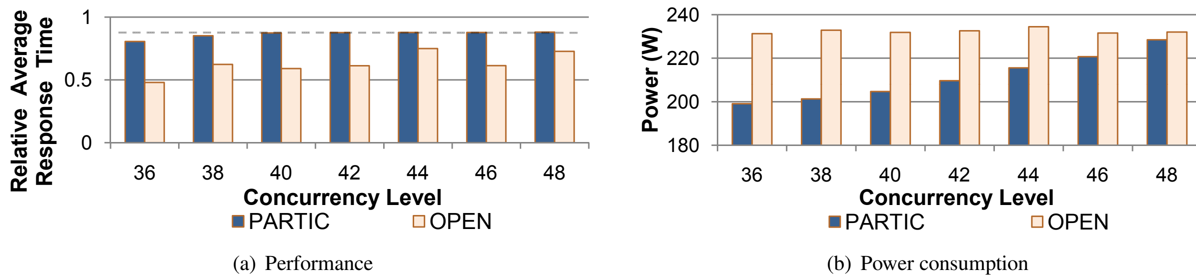
In this experiment, we demonstrate the effectiveness of integrating PARTIC with an example admission controller to handle system overload. We stress test PARTIC in a scenario where the server becomes overloaded because the workload of VM2 triples at time 900s and then returns to the original level at time 1800s. Figure 9 shows the results of PARTIC in this scenario. When the experiment begins, the response time controller dynamically changes the CPU frequency such that the average relative response time is controlled to its set point with desired power savings. At time 900s, the workload of VM2 suddenly increases threefold, causing the average relative response time to increase sharply. As a result, the response time controller detects the increase and tries to increase the CPU frequency accordingly. However, after a couple of control periods, the relative CPU frequency reaches 100% and cannot be further increased, while the average relative response time is still above the set point. At time 1017s, the system disables the response time controller and switches to the admission controller. With the relative CPU frequency fixed at 100%, the admission controller increases the rejection rate to about 0.5. As a result, approximately 50% of the web requests received by VM2 (the low-priority VM) are rejected. With admission control, the average relative response time has been sucessfully maintained at the desired level. At time 1800s, when the workload of VM2 is reduced to the original level, the av-



(a) PARTIC

(b) OPEN

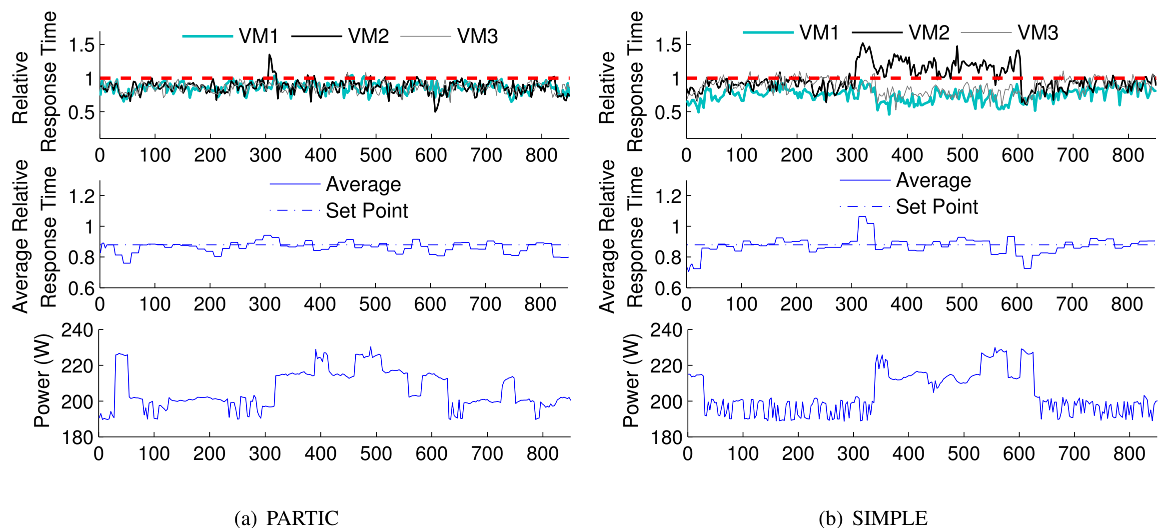
Figure 6. Performance and power efficiency comparison between PARTIC and OPEN, under a workload increase to VM2 from time 300s to time 600s.



(a) Performance

(b) Power consumption

Figure 7. Performance and power consumption with different workloads.



(a) PARTIC

(b) SIMPLE

Figure 8. Performance comparison between PARTIC and SIMPLE.

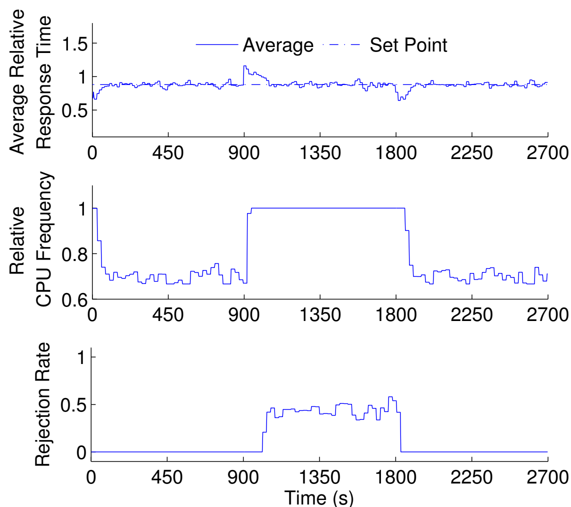


Figure 9. The integration of PARTIC with admission control to handle system overload.

erage response time becomes much shorter than the set point. The admission controller is then disabled because its desired rejection rate becomes less than 0. At the same time, the response time controller is enabled to control response time by throttling the CPU for power savings. Throughout the experiment, the average relative response times of all the VMs have been successfully kept lower than 1 most of the time. Therefore, the service-level objectives, *i.e.*, desired response times, have been guaranteed.

7 Related Work

Control theory has been applied to manage application-level performance metrics for virtualized computer servers. Padala et al. [15] propose a two-layered architecture to control resource utilization and response time. Xu et al. [16] use predictive control for dynamic resource allocation in enterprise data centers. Liu et al. [17] control the mean response time of individual applications by adjusting the percentage of allocated CPU entitlements. Zhang et al. [18] adjust the resource demands of virtual machines for fairness and efficiency. Wang et al. [19] present an adaptive controller for more robust performance. However, all the aforementioned work is not designed to reduce power consumption for virtualized servers.

Some prior work has been proposed to use power as a tool for guaranteeing required application-level SLOs in a variety of heuristic-based ways, such as using approximation algorithms [20]. More closely related to this paper, Horvath et al. [21] use dynamic voltage scaling (DVS) to control end-to-end delay in multi-tier web servers. Zhu et al. [7] develop a feedback control scheduling algorithm using DVS. Sharma et al. [5] effectively apply control theory to control application-level quality of service requirements. Chen et al. [6] present a feedback controller to manage the response time in a server cluster. Bertini et al. [22] also use SISO control to achieve energy efficiency. Although they all use control theory to manage system performance while reducing power consumption, they cannot be directly applied to virtualized computing environments because multiple application servers are correlated due to shared hardware resource. A recent work [23] discusses a heuristic solution to power control in virtualized environments. In contrast to their work, we rely on rigorous control theory for systematically developing control strategies with analytic assurance of control accuracy and system stability.

Several research projects [24, 25, 26] have successfully applied control theory to explicitly control power or cooling of computing servers. Lefurgy et al. [14] have shown that control-theoretic solution outperforms a commonly used heuristic-based solution by having more accurate power control and less overhead. Ranganathan et al. [27] propose a negotiation-based algorithm to allocate power budget to different servers in a blade enclosure for better performance. Wang et al. [28] develop a MIMO control algorithm for cluster-level power control. Those projects are different from our work because they control power consumption only and thus cannot provide guarantees for application-level performance. They also are not designed for virtualized enterprise servers.

Various admission control algorithms have been presented in related work to address server overload. Some of them adopt control theory as the foundation of their algorithm design, such as PI control [29], fuzzy control [30], and non-linear control [31]. In addition, Elnikety et al. [11] have introduced an advanced scheduling algorithm for admission control in multiple-tier e-commerce web sites. These algorithms are complementary to PARTIC and can be easily integrated into our control architecture.

Control-theoretic approaches have also been applied to other computing systems. A survey of feedback performance control in various computing systems is presented in [32]. Feedback control scheduling algorithms have been developed for real-time systems and applications [33, 34]. Control techniques have also been applied to digital control applications [35], data service and storage systems [36], and caching services [37]. While most existing work is based on simpler SISO control strategies, Diao et al. develop MIMO control algorithms to control the processor and memory utilizations for Apache web servers [38] and to achieve load balancing for database servers [39]. In our work, we develop a two-layer control architecture to provide real-time guarantees for virtualized web servers and reduce power consumption to the maximum degree.

8 Conclusions

In this paper, we have presented PARTIC, a two-layer control architecture based on well-established control theory. The primary control loop adopts a MIMO control strategy to maintain load balancing among all virtual machines so that they can have approximately the same relative response time. The secondary performance control loop then manipulates CPU frequency for power efficiency based on the uniform

relative response time achieved by the primary loop. Empirical results demonstrate that our control solution can effectively reduce server power consumption while achieving required response times for all virtual machines.

References

[1] Y. Wang, X. Wang, M. Chen, and X. Zhu, "Power-efficient response time guarantees for virtualized enterprise servers," in *Real-Time Systems Symposium*, 2008.

[2] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O devices on vmware workstation's hosted virtual machine monitor," in *USENIX Annual Technical Conference*, 2002.

[3] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles*, 2003.

[4] Microsoft Corporation, "Microsoft Virtual Server 2005," http://www.microsoft.com/windowsserversystem/virtualserver/.

[5] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu, "Power-aware QoS management in web servers," in *IEEE Real-Time Systems Symposium*, 2003.

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

[7] Y. Zhu and F. Mueller, "Feedback EDF scheduling exploiting dynamic voltage scaling," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.

[8] G. F. Franklin, J. D. Powell, and M. Workman, *Digital Control of Dynamic Systems, 3rd edition*. Addition-Wesley, 1997.

[9] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[10] P. Dorato, C. T. Abdallah, and V. Cerone, *Linear Quadratic Control: An Introduction*. Prentice Hall, 1998.

[11] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," in *International Conference on World Wide Web*, 2004.

[12] Electronic Educational Devices Inc., "Watts Up Pro Power Meter," http://www.wattsupmeters.com.

[13] "Credit Scheduler," http://wiki.xensource.com/xenwiki/CreditScheduler.

[14] C. Lefurgy, X. Wang, and M. Ware, "Server-level power control," in *IEEE International Conference on Autonomic Computing*, 2007.

[15] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[16] W. Xu, X. Zhu, S. Singhal, and Z. Wang, "Predictive control for dynamic resource allocation in enterprise data centers," in *IEEE/IFIP Network Operations and Management Symposium*, 2006.

[17] X. Liu, X. Zhu, S. Singhal, and M. Arlitt, "Adaptive entitlement control of resource containers on shared servers," in *IFIP/IEEE International Symposium on Integrated Network Management*, Nice, France, 2005.

[18] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West, "Friendly virtual machines: leveraging a feedback-control model for application adaptation," in *International Conference on Virtual Execution Environments*, 2005.

[19] Z. Wang, X. Zhu, and S. Singhal, "Utilization and SLO-based control for dynamic sizing of resource partitions." in *IFIP/IEEE International Workshop on Distributed Systems*, 2005.

[20] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo, "Multiprocessor energy-efficient scheduling with task migration considerations," in *Euromicro Conference on Real-Time Systems*, 2004.

[21] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu, "Dynamic voltage scaling in multi-tier web servers with end-to-end delay control," *IEEE Transactions on Computers*, vol. 56, no. 4, 2007.

[22] L. Bertini, J. Leite, and D. Mosse, "Siso pidf controller in an energy-efficient multi-tier web server cluster for e-commerce," in *2nd IEEE International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, 2007.

[23] R. Nathuji and K. Schwan, "Virtualpower: coordinated power management in virtualized enterprise systems," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[24] R. J. Minerick, V. W. Freeh, and P. M. Kogge, "Dynamic power management using feedback," in *Workshop on Compilers and Operating Systems for Low Power*, Sep. 2002.

[25] K. Skadron, T. Abdelzaher, and M. R. Stan, "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management," in *International Symposium on High-Performance Computer Architecture*, 2002.

[26] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No power struggles: Coordinated multi-level power management for the data center," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[27] P. Ranganathan, P. Leech, D. Irwin, and J. S. Chase, "Ensemble-level power management for dense blade servers." in *International Symposium on Computer Architecture*, 2006.

[28] X. Wang and M. Chen, "Cluster-level feedback power control for performance optimization," in *International Symposium on High-Performance Computer Architecture*, 2008.

[29] M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark, "Control-theoretic analysis of admission control mechanisms for web server systems," *World Wide Web*, vol. 11, no. 1, 2008.

[30] X. Chen, P. Mohapatra, and H. Chen, "An admission control scheme for predictable server response time for web accesses," in *International Conference on World Wide Web*. ACM, 2001.

[31] M. Kihl, A. Robertsson, and B. Wittenmark, "Analysis of admission control mechanisms using non-linear control theory," *IEEE International Symposium on Computers and Communication*, 2003.

[32] T. F. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu, "Feedback performance control in software services," *IEEE Control Systems*, vol. 23, no. 3, Jun. 2003.

[33] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *IEEE Real-Time Systems Symposium*, Dec. 2002.

[34] A. Goel, J. Walpole, and M. Shor, "Real-rate scheduling," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.

[35] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Arzen, "Feedback-feedforward scheduling of control tasks," *Real-Time Systems*, vol. 23, no. 1, pp. 25–53, Jul. 2002.

[36] M. Amirijoo, N. Chaufette, J. Hansson, S. H. Son, and S. Gunnarsson, "Generalized performance management of multi-class real-time imprecise data services." in *IEEE International Real-Time Systems Symposium*, 2005.

[37] Y. Lu, T. F. Abdelzaher, and A. Saxena, "Design, implementation, and evaluation of differentiated caching services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 5, 2004.

[38] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server," in *Network Operations and Management*, 2002.

[39] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. S. Parekh, and C. Garcia-Arellano, "Incorporating cost of control into the design of a load balancing controller," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.