Adaptive Entitlement Control of Resource Containers on Shared Servers

Xue Liu, Xiaoyun Zhu, Sharad Singhal, Martin Arlitt
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2004-178
October 14, 2004*

E-mail: xueliu@uiuc.edu, {xiaoyun.zhu, sharad.singhal, martin.arlitt}@hp.com

utility computing, resource containers, resource entitlement, adaptive control

In this paper, we describe the problem of designing online feedback control algorithms to dynamically adjust entitlement values for a resource container on a server shared by multiple applications. The goal is to quickly determine the minimum level of entitlement the container should receive in order for its hosted applications to achieve desired performance levels. Classic control theory is used as the foundation for both model identification and controller design. Specific implementation issues that affect the closed-loop system performance are discussed in detail, and a better implementation design is presented. A self-tuning adaptive controller is also presented to handle limited variations in the workload. All the controllers were implemented and evaluated on a testbed using the HP-UX PRM as the resource container technology and the Apache Web server as the hosted application inside the container. In all the experiments, our controller was able to quickly converge to the proper level of CPU entitlement to the Web server for it to track its performance target. By using our entitlement control system, shared servers can potentially reach much higher resource utilization while meeting service level objectives for the hosted applications under changing operating conditions.

* Internal Accession Date Only
Approved for External Publication

Adaptive Entitlement Control of Resource Containers on Shared Servers

Xue Liu Xiaoyun Zhu Sharad Singhal Martin Arlitt

Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304, USA
xueliu@uiuc.edu {xiaoyun.zhu, sharad.singhal, martin.arlitt}@hp.com

Abstract

In this paper, we describe the problem of designing online feedback control algorithms to dynamically adjust entitlement values for a resource container on a server shared by multiple applications. The goal is to quickly determine the minimum level of entitlement the container should receive in order for its hosted applications to achieve desired performance levels. Classic control theory is used as the foundation for both model identification and controller design. Specific implementation issues that affect the closed-loop system performance are discussed in detail, and a better implementation design is presented. A self-tuning adaptive controller is also presented to handle limited variations in the workload. All the controllers were implemented and evaluated on a testbed using the HP-UX PRM as the resource container technology and the Apache Web server as the hosted application inside the container. In all the experiments, our controller was able to quickly converge to the proper level of CPU entitlement to the Web server for it to track its performance target. By using our entitlement control system, shared servers can potentially reach much higher resource utilization while meeting service level objectives for the hosted applications under changing operating conditions.

Keywords: utility computing, resource containers, resource entitlement, adaptive control

1. Introduction

Utility computing refers to the vision of providing IT infrastructure to applications and services on demand. Enterprise grids, inspired by today's Grid technologies that serve spare compute cycles to scientific and engineering jobs, aim to serve business critical applications under the utility computing model. Enterprise applications, distinct from traditional batch-style grid workloads, have resource demands that vary over time due to changes in user needs and business conditions. These variable demands typically have high peak-to-mean ratios, which leads to low resource utilization in most of today's over-provisioned data centers with dedicated servers.

To increase resource utilization, most leading hardware and software vendors are developing virtualization technologies that allow computing resources, such as servers, networks, and storage devices, to be shared across applications. In particular, servers shared by multiple applications can potentially reach much higher resource utilization due to statistical multiplexing of demands from co-hosted applications [34].

Server consolidation is an industrial trend for reducing infrastructure and management costs and increasing return on IT investment. Current server consolidation tools typically rely on offline capacity planning to ensure that the shared server has sufficient capacity to accommodate the aggregate demand of all applications. However, this does not prevent the servers from being overloaded due to unpredictable spikes in workload. Resource contention can cause performance degradation of the hosted applications, and may result in violation of the service level agreements (SLAs) for the applications. Two classes of technologies exist on the market today to address this

problem: server partitioning and server virtualization. For instance, HP's Process Resource Manager (PRM) [13], IBM's Application Workload Manager [20], SUN's Solaris Resource Manager [35], and Aurema's ARMTech [4], are technologies for partitioning a shared server under the same operation system, and ensuring each partition's entitlement to system resources, such as CPU, memory, and disk bandwidth, under overload conditions. On the other hand, server virtualization technologies [6][15][31][36] allow multiple virtual servers to be created on the same physical machine and encapsulate resource demands of applications inside different virtual servers. In this paper, we do not distinguish between these two technologies, and use the general term "resource container" to refer to a partition of a server that has certain entitlement to shared resources on the server.

Resource containers are useful for performance isolation and service differentiation for applications on shared servers. However, the benefit of statistical multiplexing cannot be fully utilized if resource entitlements of the containers are predetermined statically. System resources on the shared server should be provided to hosted applications on an as-needed basis. For resource containers that host applications with SLA-based performance requirements, we need to guarantee that the container is always provided with enough resources such that the performance goals can be met. At the same time, over-provisioning of resources should be prevented so that more applications can be hosted on the same server. So the key question is, what is the minimum amount of system resource an application needs in order to meet its performance objective? This problem can be solved effectively using a feedback control approach, as illustrated in Figure 1. A controller periodically takes performance measurements of the application from a monitoring agent, compares it with the desired performance, and adjusts entitlement values for the resource container to meet the application's performance goal. The changes to the entitlement can be effected through either exposed APIs or configuration utilities provided by the container. The performance monitoring agent can either be monitoring APIs provided by the application, or some standalone program that computes performance metrics from appropriate log files.

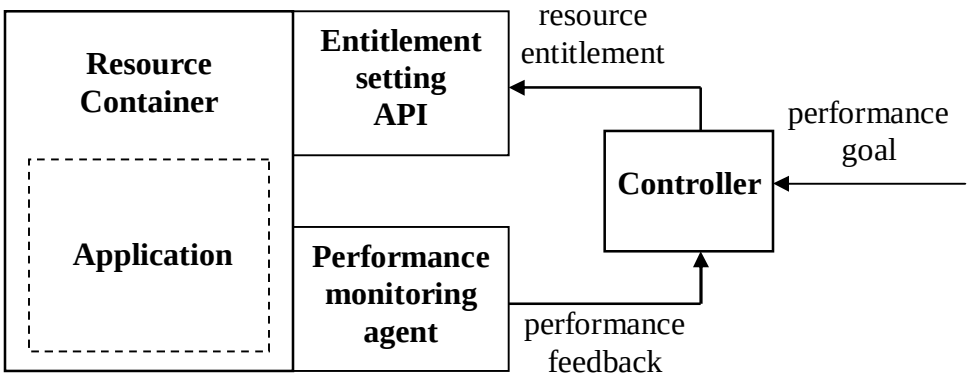


Figure 1. Entitlement control for a resource container using feedback

Most existing technologies for realizing this feedback loop online rely on policies or heuristics [20][29][36]. Heuristic algorithms are usually simple and easy to implement. However, they require a great deal of expert knowledge, therefore are typically domain specific. In addition, they do not provide stability guarantees, and may lead to large oscillations in certain metrics. In this paper, we propose a more systematic approach that uses control theory as the foundation for designing the control algorithms for the feedback loop in Figure 1. Control theory offers a great analytic engine and mature methodology for system modeling, analysis, design, and simulation. The rich theory provides useful guidelines for analyzing systems' stability, performance, as well as tradeoffs between the two. For systems that cannot be easily described by first principles, the system identification process utilizes a black-box approach to infer a system model from

measurement data. Moreover, control theory is especially useful in an uncertain environment where system conditions and application requirements cannot be predicted accurately in advance, which is typically true for systems shared by enterprise applications. Finally, due to the use of short-term dynamic models to predict system behavior, this approach can quickly react to changes in workload intensity or system conditions automatically, in complement to offline planning and policy-based online adjustments that typically work at longer time scales.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the architecture of the entitlement control system and the set up of our testbed in a case study. Section 4 demonstrates how the system model can be inferred from experimental data using standard system identification techniques in control theory. Section 5 describes offline design of a PI controller under a fixed workload, and discusses implementation issues and resolutions. Section 6 introduces the design of an adaptive controller and its online operation. Section 7 presents performance evaluation results for both the fixed controller and the adaptive controller. Finally, Section 8 offers conclusions and discusses future research directions.

2. Related Work

The concept of a *Resource Container* [5] was first proposed as a new operating system abstraction which separates the notion of a protection domain from that of a resource principal. Resource containers enable fine-grained resource management in server systems and allow the development of robust servers, with simple and firm control over traditional priority policies. Our notion of resource container is broader in the sense that it is agnostic about the specific technology used to create the resource container and how resource entitlement is enforced inside the container. In addition, the implementation of the resource container in [5] requires modification of both the kernel and the server applications, while our entitlement control system relies on existing server partitioning or virtualization technologies and externally exposed APIs or simple configuration utilities for the containers, and does not require a change of the applications running inside the container. Our goal is to provide a non-intrusive mechanism for dynamically controlling the entitlement of any given resource container.

There have been various research efforts in providing resource reservation, monitoring, and enforcement capabilities on shared servers. The *Rialto* operating system [16] offers support for scheduling multiple independent real-time applications along with traditional timesharing applications on the same machine using a combination of CPU reservations and time constraints. *Processor Capacity Reserves* [30] is a scheduling framework that supports reservation and admission control for multimedia applications. The reserve abstraction, though specifically designed for the microkernel architecture, can measure and control processor usage by individual applications. *Resource Kernels* [33] is real-time kernel that provides applications with explicit, timely, guaranteed and protected access to system resources. Under the support of resource kernels, an application can request the reservation of a specified amount of a resource, and the kernel can guarantee that the requested amount is exclusively available to that application. These resource reservation frameworks require that the entitlement of an application's access to system resources be determined a priori. This is not appropriate for enterprise applications whose exact resource demands are unknown in advance, and typically fluctuate over time. Reservation for peak load is not desirable because it would lead to low resource utilization. In this paper, we focus on designing feedback control algorithms that dynamically adjust an application's entitlement to resources based on its real-time resource needs during execution in order to meet its performance goals.

Some vendors of server partitioning or virtualization technologies provide additional application/workload management tools for controlling application performance on shared servers using feedback. The *IBM z/OS workload manager* [21] for its zSeries servers lets the user of the system define performance goals for each application and assigns importance to each goal,

then it automatically figures out how much CPU and storage resource should be given to each application to meet the goal while constantly monitoring its performance. The *HP-UX Workload Manager* [14] allows the definition of a service level objective (SLO) for a workload, and implements a proportional (P) controller to dynamically adjust resource allocation to each workload to achieve its SLO [29]. The algorithm contains a number of parameters that the user needs to tune, mostly by trial-and-error, to achieve optimal performance of the controller. In this paper, we use control theory for designing such feedback control algorithms.

Feedback control theory has had a long history of application in many engineering domains. In the past several years, with the help of digital control technology [11], control theory has been successfully applied to performance or quality of service (QoS) control of a variety of computer systems and software, including Lotus Notes email server [9][12], Apache Web server [1][7][8][10][23], Squid proxy server [25][26], Lustre file system [18], as well as a 3-tier e-commerce site [19]. The metrics being controlled could be system-level metrics, such as CPU and memory utilization [1][8][10][24], cache hit ratio [25][26], server queue length [7][12], and server power consumption [17], or application level metrics such as response time and/or throughput [18][19][23], or even business level metrics such as profits [9]. The control mechanisms for these systems include admission control or request throttling [7][18][19], Web content adaptation [1], tuning of application configuration parameters [8][9][10][12], resource allocation [23][25][26], and frequency scaling [17]. The types of control algorithms used include integral (I) control [12], proportional and integral (PI) control [1][7], pole placement [8], linear quadratic regulator (LQR) [8][10], fuzzy control [9], model predictive control (MPC) [17][24], and adaptive control [18][19][25]. Adaptive control, in particular, has received much interest lately due to its self-tuning capability that allows the controller to adapt to changes in operating conditions and workloads automatically.

Our work is distinguished from the prior work in that we propose a generic approach for dynamically controlling resource entitlements of server applications that is built upon state-of-the-art server partitioning or virtualization technologies and existing application capabilities. Although our experimental study used the HP-UX PRM and the Apache Web server as examples, the control system architecture we proposed can be applied to any resource container technologies and any applications hosted inside such containers on shared servers. Furthermore, we designed and implemented an adaptive controller that self-tunes its parameters based on online estimates of the system model and validated its effectiveness against variation in workload on a real system testbed.

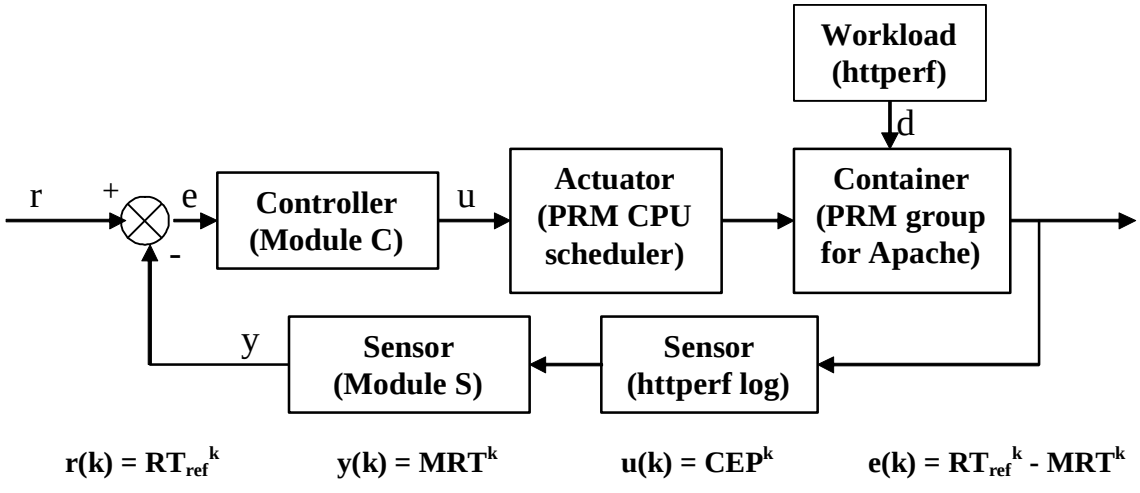


Figure 2. Block diagram of the resource entitlement control system

3. Control System Architecture and Testbed Setup

In this section, we introduce a case study that uses the HP-UX Process Resource Manager (PRM) [13] as an example of a resource container technology and the Apache Web server [2] as an example of an application running inside a resource container. Figure 2 illustrates the block diagram of the entitlement control system in our study. The subsections that follow explain in detail each block in the diagram.

3.1 Server Application

We chose the Apache Web server as the application to be hosted inside a resource container. The recent Apache release 2.0.48 was used. The Apache 2.x releases use a thread model called the Multi-Processing Module (MPM) to improve the performance over the Apache 1.3.x releases that used a process model. The new MPM implements a hybrid multi-process multi-threaded server. By using threads to serve requests, it is able to serve a large number of requests with less system resources than the previous process-based server. At the same time, Apache 2.X retains much of the stability of a process-based server by keeping multiple processes available, each with a variable number of threads to serve incoming requests.

3.2 Resource Container and Actuator

In our case study, we use HP-UX PRM as the resource container technology and the actuator for our control system. PRM is a resource management tool that allows system administrators to fine-tune how system resources such as CPU, physical memory, and disk bandwidth on a server are shared by multiple users or applications. PRM controls the allocation of these resources to PRM groups. Each PRM group is a conceptual partition of the system's resources, therefore a resource container. Because this partitioning is accomplished in the operating system, it can be changed at any time, even while the system is in use. During system overload conditions, PRM guarantees a minimum entitlement to system resources by each PRM group. Optionally, if CPU or memory capping is enabled on a PRM group, PRM ensures the group's usage of CPU or memory does not exceed the cap regardless of whether the system is fully utilized.

In this paper, we focus on CPU as the key system resource for applications, and rely on the CPU scheduler in PRM to enforce the CPU entitlement setting for a PRM group. We define the CPU entitlement percentage, CEP (u), as the entitlement to a percentage of CPU cycles used by all the Apache processes belong to the same PRM group.

3.3 Workload Generator and Sensor

We use httperf[1] [32], a scalable client workload generator, to continuously send HTTP requests to the Apache Web server. We chose client-perceived mean response time, MRT, as the performance metric for the Web server. We modified httperf 0.8 to log the response time of every request. The resulting httperf log serves as our first sensor module for application performance (see Figure 2). We have programmed another sensor module S to compute the MRT of all the requests that returned during each sampling interval. The sampled MRT is fed into the controller for calculating control actions for the next interval. Because the workload mix and intensity affect the degree of how system resources inside the resource container are stressed, the workload also serves as a disturbance (d) to the control system.

3.4 Controller

The goal of the controller is to compute the proper level of CPU entitlement for the Web server to maintain the measured MRT around the desired response time. The latter is referred to as the

[1] ftp://ftp.hpl.hp.com/pub/httperf

reference signal in control theory, therefore denoted by RT_{ref}. The settings for RT_{ref} can be based on the SLA between the service/application provider and its users. At every sampling instant k, the measured MRT^k (y(k)) for the previous sampling interval is compared to the current reference RT_{ref}^k (r(k)), and the difference e(k) is fed into the controller module C to compute the new CPU entitlement value, CEP^k (u(k)), for the current interval. The above logic is implemented in a controller module C. The output of the controller module is subsequently fed into the actuator, i.e., PRM's CPU scheduler, to reallocate the CPU cycles during this sampling interval.

3.5 Testbed Setup

All experiments including system identification and control were conducted on a testbed of two computers connected by 100 Mbps Ethernet, illustrated in Figure 3. The client machine runs httperf, the sensor module S, and the controller module C. The client machine has a 500 MHz Pentium III processor and 512 MB RAM. It runs Red Hat Linux 7.3 with kernel version 2.4.18. The server machine is used to run the Apache Web server 2.0.48 on HP-UX B11.00. It is an HP9000-R server with one 180 MHz PA-8000 processor and 512 MB RAM. The experimental setup is as follows.

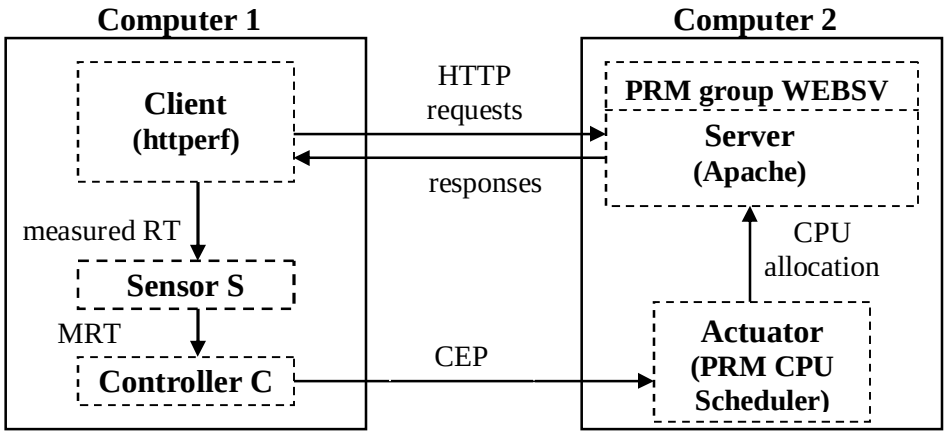


Figure 3. Set up of the entitlement control testbed

- **Client workload**: Two sets of workloads were generated and tested in our experiments. For both sets, only static content was used. This prevents the system memory from becoming a bottleneck. In addition, the total size of the working file set is small enough to fit in the file system buffer cache so that no stress on the disk bandwidth is created either. The first set, WL1, consists of a sequence of static HTTP requests for the same URL, but at a different rate of requests per second. The second set, WL2, uses a working set of 540 distinct files that have sizes ranging from 1KB to 90 KB. In WL2, We used the '–wsesslog' option of httperf to generate the workload. The corresponding session log file was generated in such a way that each session fetches a file that is randomly chosen out of the 540 files with a burst length of 10 requests per connection. The session rate is exponentially distributed with a mean of 30 sessions per second. The purpose of the second set of workloads will be explained in the performance evaluation section.
- **Web server:** During all the experiments, we use the default Apache server settings and configuration parameters recommended by the Apache group.
- **PRM:** We use the version of PRM inside the HP-UX B11.00 kernel. The PRM configuration file */etc/prmconf* is modified every time a new CPU entitlement value is calculated, and the PRM utility *prmconfig* is used to reconfigure the entitlement settings of all the PRM groups. For the purpose of our experiment, all the Apache related users and processes are placed in

one PRM group called "WEBSV". This serves as the resource container for the Web server application. Since CPU capping is enabled, the CPU entitlement percentage CEP also acts as an upper bound on the amount of CPU cycles used by the WEBSV group, enforced by the PRM CPU scheduler. In addition, the real CPU utilization by the WEBSV group is measured every second using the *prmmonitor* command provided by PRM and recorded in a log file that can be analyzed after each experiment.

- **Sensor and controller:** Both the sensor module S and the controller module C are implemented in Perl. They can be placed on the client machine, the server machine, or a third machine. For the ease of parsing the httperf log, we placed them on the client machine.

4. Model Identification

In this section, we establish the open-loop dynamic model for the mapping between the WEBSV PRM group's CPU entitlement percentage (CEP^k) and the mean response time (MRT^k). For computer systems such as a Web server, it is difficult to obtain the model using first-principles due to the complex nature of the system. Therefore, we treat it as a black-box and infer the model from externally observable metrics using standard system identification techniques [22].

A single-input-single-output (SISO) model is used, where the input of the system $u(k)= CEP^k$, and the output $y(k)=1/MRT^k$ for each sampling interval. The reason why the inverse of MRT is used as the output is that MRT is roughly inversely proportional to the Web server's throughput, while the latter is proportional to the CPU entitlement. Therefore, we expect that the MRT is inversely proportional to the CEP. Using the inverse of MRT allows us to find a simple linear model as the basis for the controller design.

For the system identification experiments, we used the simple fixed workload WL1 at a rate between 300 and 600 requests/sec to stress the CPU of the server at a certain level. For full excitation of the system, we varied CEP^k using a pseudo-random sequence uniformly distributed in the interval [CPULOW, CPUHIGH]. We set CPULOW=20% and CPUHIGH=90% as the minimum and maximum CPU entitlement settings used for both system identification and control. At the beginning of each sampling interval *k*, u(k) was randomly chosen and fed into PRM for bounding the WEBSV group's CPU utilization. The inverse of MRT, y(k), was measured by the sensor module S from the httperf log. The experiment lasted 30 minutes, resulting in a time series of 120 input-output pairs (u(k), y(k)). We used the first 60 samples for identification, and the remaining 60 samples for validation. The experiment was repeated at different settings of the request rate, with different sampling intervals.

After experimenting with various linear parametric models using the Matlab *System Identification Toolbox* [28], we have two observations. First, a longer sampling interval leads to a better fit of the model in general, while a shorter sampling interval produces data that is too noisy to be fit using a simple linear model. On the other hand, the controller may be too slow when the sampling interval is too long. We chose a sampling interval of 15 seconds as the result of the tradeoff. The second observation is that the following first-order auto-regressive (AR) model [22] provides reasonably good prediction for the inverse of MRT for all request rates.

$$y(k+1) = b_0 u(k) + a_1 y(k). \qquad (1)$$

The corresponding z-transform of the transfer function from u(k) to y(k) follows.

$$G(z) = \frac{Y(z)}{U(z)} = \frac{b_0}{z - a_1} = \frac{b_0 z^{-1}}{1 - a_1 z^{-1}}. \qquad (2)$$

Notice that the system has a one-sample delay, z^{-1}, in the response of y(k) from u(k). There are four potential contributing factors to this delay. First, the Web server itself may have a

delayed change in the response time when the CPU allocation to it changes; Second, the output, y(k), is an average measure, which incurs up to one sampling interval of delay from the real response time; Third, it takes time for the CEP^k command to be sent from the client machine to the server and be written into the PRM configuration file; Fourth, the actuator (PRM's CPU scheduler), which enforces the real CPU utilization of the WEBSV group to obey the entitlement value, requires some time for the enforcement to take effect.

The least-squares based methods [22] in the *System Identification Toolbox* were used to estimate the first-order AR model. In addition to visually inspecting the accuracy of the prediction, one goodness-of-fit measure, r^2, was used to compare models with different parameter values. r^2 denotes the percentage of the variation in y(k) accounted for by the model. For the rate of 600 requests/sec, we arrived at the following parameters: $b_0 = 0.17$, and $a_1 = 0.46$. The resulting open-loop transfer function is

$$G(z) = \frac{0.17}{z - 0.46}. \qquad (3)$$

The corresponding r^2 value is 50.3%. Figure 4 shows the comparison of the measured output and simulated output (both inverse of MRT) from the above model on the validation data. We notice that although the model prediction misses some of the peak values in the real data, it is able to capture most of the fluctuations with reasonable accuracy. The results for other request rates are similar, while leading to different values of b_0 and a_1.

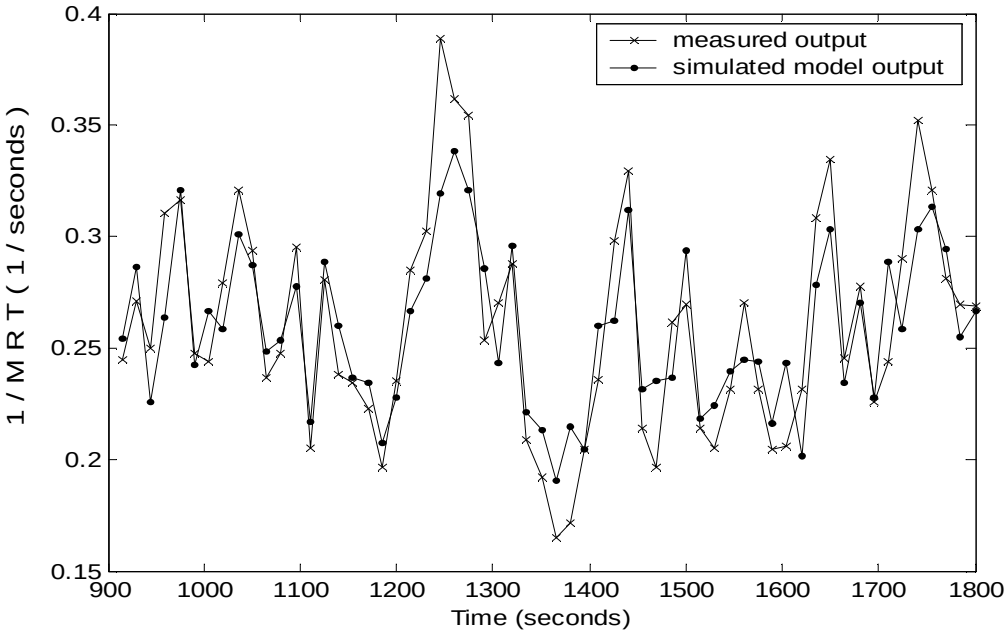


Figure 4. Comparison of measured output vs. model prediction

5. Offline Controller Design and Implementation

We now apply control theory to design the CPU entitlement controller in the feedback loop. The problem we study here is referred to as a regulation problem in the control literature, where the controller aims to maintain the MRT around a given target value by dynamically adjusting the WEBSV group's CPU entitlement at every sampling interval.

Table 1 lists the desired properties of the closed-loop control system along with the corresponding design goals for the controller.

Table 1. Real system desired properties vs. controller design goals

Desired System Properties	Controller Design Criteria
Stable	Poles within the unit-circle
Reference tracking	Zero steady-state error
Fast response	Small rise time (t_r) and settling time (t_s)

For our specific control system, stability means minimum oscillation in the controlled metric, i.e., MRT; in control theory, this requires that the poles of the closed-loop transfer function be within the unit-circle. The ability to track reference means the measured MRT eventually converges to the given response time target; in control theory, this is referred to as zero steady-state error. Finally, fast response means when the response time reference changes, the measured MRT should be able to track the change in a timely manner; in control theory, we use the rise time t_r and the settling time t_s to measure the speed of the response, where the rise time refers to the time it takes to reach 90% of the reference value, and the settling time refers to the time it takes to settle within $\pm 5\%$ of the reference value. These criteria are used as guidelines when we determine the controller algorithm and parameters.

We chose Proportional-Integral (PI) controller for its simplicity and the ability to achieve zero steady-state error. The PI controller for a discrete-time system has the following form:

$$u(k) = u(k-1) + (K_p + K_i)e(k) - K_p e(k-1), \quad (4)$$

where $e(k)=r(k)-y(k)$ is the error between the reference value and the measured output. For our system, we have $e(k) = 1/RT_{ref}^k - 1/MRT^k$. The parameters used were $K_p = 2.3$ and $K_i=2.7$. This corresponds to the following transfer function for the controller:

$$K(z) = \frac{U(z)}{E(z)} = 5 \times \frac{z - 0.46}{z - 1}. \quad (5)$$

Analytical calculation shows that the resulting closed-loop system should be able to track the references at steady state, and has both rise time and settling time at 30 seconds, corresponding to 2 sampling intervals.

In the following subsections, we describe three controller implementation designs and discuss their advantages and disadvantages. In all three implementations, the control interval is set as the same as the sampling interval, $T=15$ sec.

5.1 Controller Implementation Design 1

Figure 5 illustrates a typical timeline of implementing the above PI controller. In this design, we use a real-time timer to kick off the controller every T seconds. Starting from the beginning of each control interval T_k, the following two steps are performed:

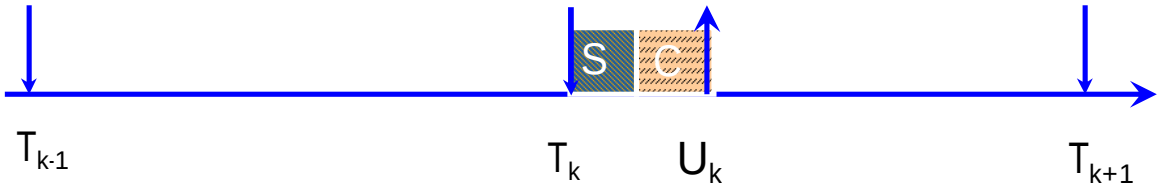


Figure 5. Controller implementation design 1

Step 1: The sensor module S parses the httperf log on the client machine and computes the mean response time MRT^k of all the requests that were completed in the last control interval $[T_{k-1}, T_k)$.
Step 2: The controller module C compares the inverse of MRT^k with the inverse of the response time reference RT_{ref}^k, and uses the error to compute the control input, CEP^k, for the current

control interval. After this is done, the control input is communicated to the server machine at time U_k, and will be subsequently actuated by PRM.

This controller implementation is intuitive. However, since httperf uses buffered stream I/O to log the requests' response time measurements, if at each timer expiration time T_k, we immediately begin parsing the httperf log file, the response time measurements of some of the requests have not yet been logged. This means MRT^k will be calculated from an incomplete sample of the responses, which can degrade the performance of a traditional control system. We observed similar performance degradation of the control system in our own testbed, which is demonstrated in Section 6.

One naïve remedy for this problem is to turn off the streamed I/O in httperf. However, due to the large volume of requests seen by a typical Web server, using non-streamed I/O will greatly affect the performance of httperf. We could also modify the httperf source code so that the logging is synchronized with our sampling interval, or even better, implement both the sensor module S and the controller module C inside httperf. However, this becomes an intrusive approach and is not generic enough for it to be applied to all applications. Thus we prefer an alternative solution to this problem.

5.2 Controller Implementation Design 2

To solve the incomplete sample data problem in controller implementation design 1, we propose the controller implementation design 2 as shown in Figure 6.

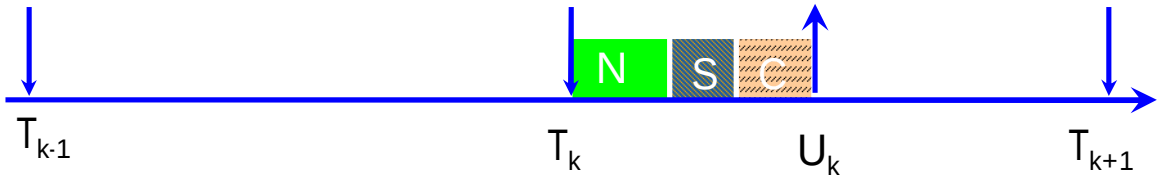


Figure 6. Controller implementation design 2

In this new design, at each timer expiration time T_k, instead of parsing the httperf log immediately, we add a nap time (denoted by block N) to wait for httperf to finish logging the response time measurements of all the requests that are completed in the previous interval $[T_{k-1}, T_k]$. The sensor module S and controller module C follow afterwards as in design 1.

The time to nap is a design parameter that should be chosen carefully. Having a nap time that is too short will not accomplish the goal of obtaining complete samples. On the other hand, the nap time should not be too long so that the whole sequence of control actions can be completed by the deadline (beginning of the next interval T_{k+1}). In our testbed, after carefully measuring the timing requirements of each module and the streamed I/O of httperf, we found that the nap time of 3 seconds is a good choice, as all requests were logged and no deadlines were missed.

However, in controller implementation design 2, there is another deficiency that may degrade the performance of the closed-loop system. In a control system, the irregularity of control action times is referred to as control jitter. Control jitter comes from the variable execution times of different modules in each controller instance. This is illustrated in Figure 7. In control interval $[T_{k-1}, T_k]$, the control action time is U_{k-1}. Since timer expiration time T_{k-1} has the highest timing regularity, we use $t_{k-1}=U_{k-1}- T_{k-1}$ to denote the relative control action time for interval $[T_{k-1}, T_k]$. Similarly, $t_k=U_k- T_k$ is the relative control action time for interval $[T_k, T_{k+1}]$. We see from Figure 7 that due to the possible variable execution times of the sensor and the controller modules, there may be some jitter between t_{k-1} and t_k.

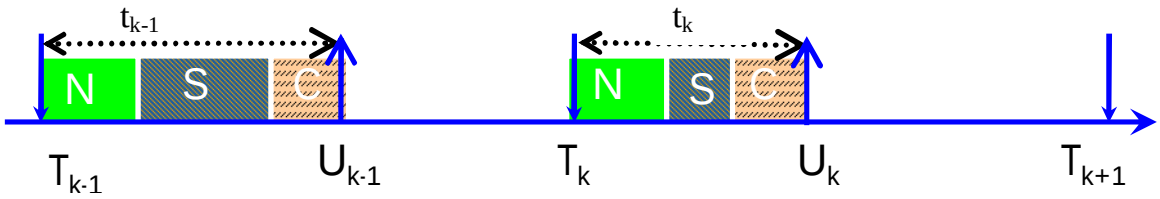


Figure 7. Illustration of jitter in controller implementation design 2

It is well known that the presence of jitter in a controller's execution will degrade the performance of the system, even causing a critical failure of the system [27]. To minimize the impact of jitter, we propose a third controller implementation design.

5.3 Controller Implementation Design 3

In controller implementation design 3, we address the control jitter problem by placing the control action time as close to T_k as possible in each control interval. We achieve this by withholding the calculated control input u(k-1) for the control interval [T_{k-1}, T_k] until the beginning of the next interval, T_k. The illustration is shown in Figure 8. By making U_{k-1} almost the same as T_k, we minimize the control jitter by removing the impact of variable execution times from the three modules (N+S+C).

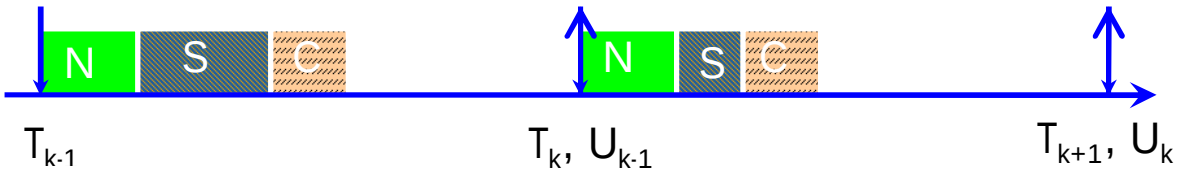


Figure 8. Controller implementation design 3

In order to use controller implementation design 3, we need to revise the controller we had designed at the beginning of this section, as holding the control input to the next interval introduces an extra delay of one sampling interval. This results in the following controller algorithm:

$$u(k+1) = u(k) + (K_p + K_i)e(k) - K_p e(k-1). \qquad (6)$$

By choosing $K_p = 0.8$ and $K_i = 0.95$ based on our design criteria, we arrived at the following transfer function for the controller:

$$C(z) = \frac{U(z)}{E(z)} = 1.75 \times \frac{z - 0.46}{z(z-1)}. \qquad (7)$$

Analytical calculation shows that the closed-loop system has a rise time of 45 seconds (3 samples) and a settling time of 90 seconds (6 samples). Notice that both the rise time and the settling time are longer than the controller design with no delay, which is to be expected since a smaller gain is chosen for the controller in exchange for stability.

6. Adaptive Controller Design

The previous section described how a PI controller can be designed offline based on the dynamic input-output model obtained through system identification experiments. However, the system identification experiments were conducted on the simple fixed workload WL1. As a consequence, the offline controller is designed specifically to meet the resource needs of that particular workload. In a real system, workload conditions can be viewed as a disturbance on the controlled

system. In general, it is very difficult to find a single linear model that characterizes the system's behavior under all workload conditions. As the workload changes, the best model to describe the system's behavior changes accordingly. This calls for the design of an adaptive controller, which repeatedly estimates the dynamic model based on online input-output measurements, and computes controller parameters online based on the current estimated model.

For our entitlement control testbed, a simple indirect self-tuning regulator [3] was implemented. One assumption we made was in spite of changes in the workload condition, a first-order AR model was always appropriate for predicting the MRT for the next sample interval. This means, the input-output model has a fixed structure, but the parameters are time-varying depending on the current workload in the system. Therefore, equation (1) remains as the open-loop system's model, and the PI controller with one-step delay can still be used to regulate the Web server's MRT. Figure 9 shows a block diagram for the self-tuning regulator, which is a variation of the general one in [3], but contains specific information for our testbed.

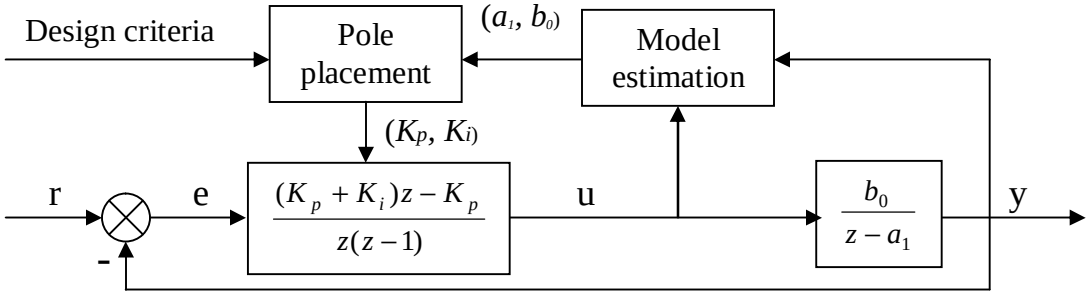


Figure 9. Block diagram of an indirect self-tuning regulator

We use the recursive least-squares (RLS) method [3] to estimate the two parameters b_0 and a_1 for the first-order AR model. The updated parameter values at every control interval are fed into a pole placement module. The pole placement module chooses the desired closed-loop system poles based on the design criteria, such as stability, rise time, settling time, and bound on overshoot, and then computes the appropriate values for the K_p and K_i gains in the PI controller. The controller module implements the PI algorithm using the calculated gain values.

7. Performance Evaluation

In this section, we present the experimental results for the various controllers that we described in earlier sections, and offer a comparison with respect to the closed-loop system performance.

First, we used the fixed workload WL1 with a rate varying between 300 and 600 requests/second the same as what we used for system identification, to test the three implementations of the offline controller. To avoid the starting phase of httperf, the controller was turned on at 60 seconds after httperf started, and the test lasted 120 sampling intervals (1800 seconds). In order to test how well and how fast the closed-loop control system can track changes in the response time target RT_{ref}, we set RT_{ref} at 3, 5, 3 seconds during time intervals [0s-600s), [600s-1200s), [1200s-1800s), respectively.

Figure 10 shows the performance of the closed-loop system for workload WL1 at rate=600 requests/second under controller implementation design 3. The middle graph (b) shows the measured MRT and the RT_{ref} as a function of time. With a small amount of delay (within 3-8 sampling intervals), the controller could track changes in RT_{ref} and maintain the MRT within $\pm 15\%$ of RT_{ref}. Tracking was achieved by dynamically adjusting the CPU entitlement for the WEBSV group at every control interval (15 second). To see this, the top graph (a) shows the control input CEP^k and the measured CPU utilization used by the WEBSV group during the experimentation. The small gap between the two curves reflects both the communication delay

from the client machine to the server machine and the inherent error (and delay) in the actuator, the PRM CPU scheduler. (Note that this gap has been taken into account in our model since the input data we used for system identification was CEP^k, not the real CPU utilization.)

As we can see, for each setting of MRT target, our controller was able to quickly determine what is the correct level of CPU entitlement the WEBSV group should receive in order to meet that target. For example, a MRT target of 3 seconds requires a CPU entitlement of about 70-80%, while an MRT target of 5 seconds requires a lower entitlement setting at around 40-50%. The feedback controller carried out these adjustments automatically.

The bottom graph (c) shows the mean throughput of the server in the number of requests completed in each control interval. As the MRT increases, the mean throughput decreases as expected. As a result, if the SLA is with respect to throughput instead of response time, then a higher throughput target would require a higher level of CPU entitlement for the WEBSV group. For example, a CPU entitlement of 40-50% produces a throughput of about 200 requests/second, while a CPU entitlement of 70-80% produces a throughout of about 300 requests/second.

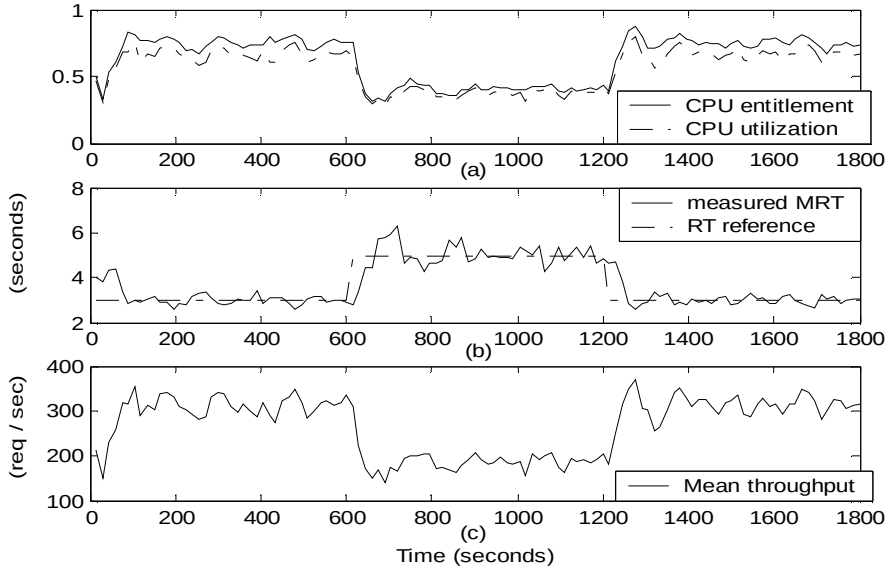
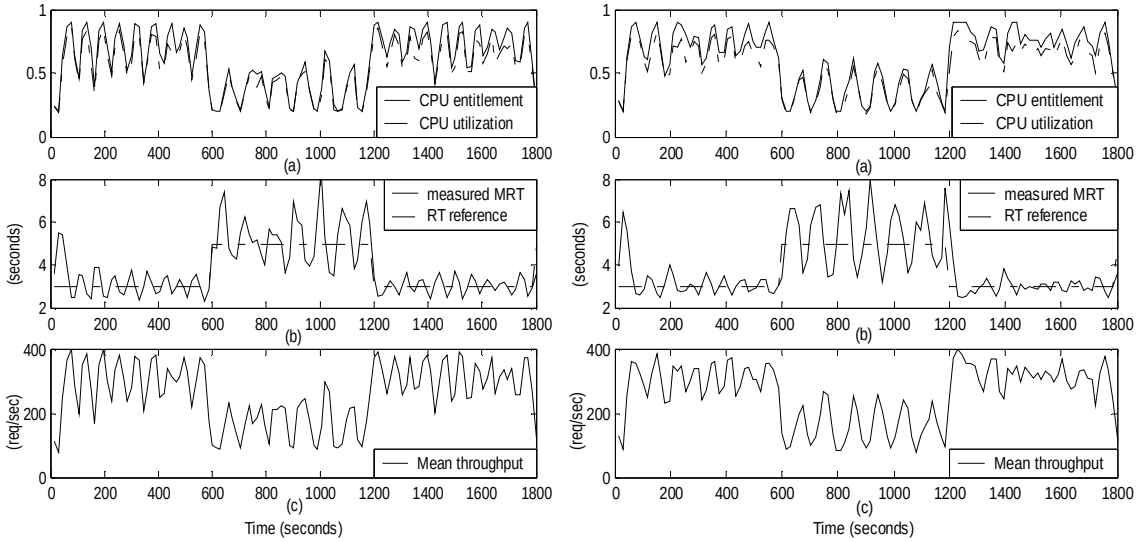


Figure 10. Closed-loop system performance under controller implementation design 3

We also experimented with the controller implementation designs 1 and 2 discussed in Section 5. The results are shown in Figure 11. Compared with controller implementation design 3, these two controller implementations are much less stable and result in large oscillations in response time, throughput, as well as CPU utilization of the WEBSV group.

Finally, we tested the self-tuning adaptive controller against the variable workload WL2. Figure 12 shows the result for a mean request rate = 30 sessions/second (with an exponential distribution). In the earlier sections, we discussed how different rate settings for even the simpler workload WL1 would lead to different parameter values for the model, which means a single fixed controller may not work well for all the rate settings. Since httperf does not allow the rate variable to be changed in the middle of a run, we chose to use variation in the file size in this workload to mimic the change in the CPU demand the workload places on the server during a single experiment. As we can see, for each fixed MRT target, the adaptive controller was able to maintain the MRT within $\pm$ 20% of the target in spite of the variation in the workload. When the MRT target changed over time, it was able to track the changes within 5-9 sampling intervals and arrive at the proper levels of CPU entitlement for different target settings. However, the measured MRT does converge slower to the target value and has relatively more oscillations compared to the single controller specifically designed for a given workload. This is due to online estimation

of the parameters. It seems to be a reasonable tradeoff considering the larger classes of workload the adaptive controller is able to handle.



(i) implementation design 1 **(ii) implementation design 2**

Figure 11. Closed-loop performance under controller implementation design 1 and 2

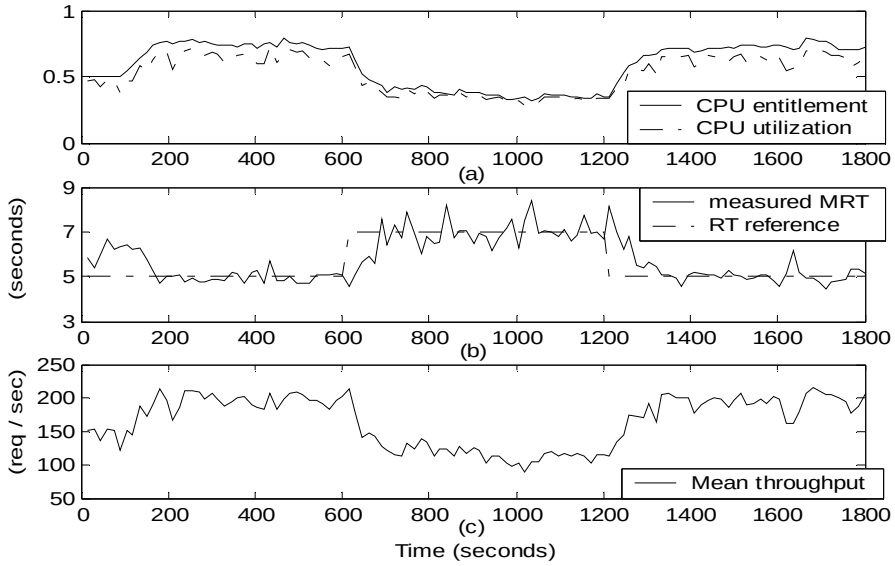


Figure 12. Closed-loop performance of the adaptive controller

8. Conclusions and Future Work

In this paper, we described the problem of designing online feedback control algorithms to dynamically adjust entitlement values to a resource container on a shared server. A PI controller was designed offline for a fixed workload and a self-tuning adaptive controller was described to handle limited variations in workloads. All the controllers were implemented and tested on our real-time control testbed using the HP-UX PRM as the actuator and the Apache Web server as the hosted application inside the resource container. Both controllers were able to quickly converge to the minimum level of entitlement the container should receive in order for its hosted applications to achieve their performance goals. This technique is particularly useful when

multiple applications are co-hosted on the same server, which is the key feature of today's server consolidation practices and utility computing environments. By using our entitlement control system, shared servers can reach much higher resource utilization while meeting service level objectives for the hosted applications. In future work, it will be interesting to evaluate the scenario where multiple applications are competing for resources on the shared server, and not all applications' performance goals can be met at the same time. In this case, policy-based or utility-driven prioritization schemes may be incorporated so that entitlement values for individual applications can be decided in a way that maximizes the overall benefit generated from the server.

As ongoing work, we are including dynamic content into our workload set so that we can stress the memory of the resource container along with the CPU, and include memory entitlement as another control input in our system. Further explorations can allow disk bandwidth entitlement and system throughput to be included into the problem in a similar fashion. This would result in a multiple-input-multiple-output (MIMO) model for the controlled system, which requires that a broader set of controller design methodologies be investigated.

Another interesting direction is to apply the same approach to environments with virtual machine technologies. This would rely on the power and flexibility of the entitlement setting APIs exposed by various products. We intend to determine if adding more dynamic control and self-tuning capabilities on top of the server virtualization technologies would enable enterprise grids to achieve much higher utilization in the infrastructure, better predictability in application performance, and quicker adaptation to changes in business priorities and user demands.

References

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

[2] Apache Web server, http://www.apache.org/

[3] K.J. Astrom and B. Wittenmark, *Adaptive Control (2nd Edition)*, Prentice Hall, 1994.

[4] Aurema ARMTech, http://www.aurema.com/

[5] G. Banga, P. Druschel, and J.C. Mogul, "Resource containers: A new facility for resource management in server systems," *3rd USENIX Symposium on Operating Systems Design and Implementation*, February, 1999.

[6] P. Barham, *et al.*, "Xen and the art of virtualization," *19th ACM Symposium on Operating Systems Principles*, October, 2003.

[7] P. Bhoj, S Ramanathan, and S. Singhal, "Web2K: Bringing QoS to Web servers," *HP Labs Technical Report*, HPL-2000-61, May 2000.

[8] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury, "MIMO control of an Apache Web server: Modeling and controller design," *American Control Conference*, 2002.

[9] Y. Diao, J.L. Hellerstein, S. Parekh, "Using fuzzy control to maximize profits in service level management," *IBM Systems Journal*, Vol. 41, No. 3, 2002.

[10] Y. Diao, J.L. Hellerstein, S. Parekh, and J.P. Bigus, "Managing Web server performance with AutoTune agents," *IBM Systems Journal*, Vol. 42, No. 1, 2003.

[11] G.F. Franklin, D.J. Powell. and M.L. Workman, *Digital Control of Dynamic Systems (3rd Edition)*, Prentice Hall, 1997.

[12] N. Gandhi, S. Parekh, J. Hellerstein, and D.M. Tilbury, "Feedback control of a Lotus Notes server: modeling and control design," *American Control Conference*, 2001.

[13] HP Process Resource Manager, http://h30081.www3.hp.com/products/prm/index.html

[14] HP-UX Workload Manager, http://h30081.www3.hp.com/products/wlm/index.html

[15] HP Virtual Server Environment for HP-UX, http://h71028.www7.hp.com/enterprise/cache/10381-0-0-0-121.aspx

[16] M.B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," *16th ACM Symposium on Operating Systems Principles*, October, 1997.
[17] N. Kandasamy, S.Abdelwahed, and J.P. Hayes, "Self-Optimization in computer systems via online control: Application to power management," *International Conference on Autonomic Computing*, May 2004.
[18] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance isolation and differentiation for storage systems," *12th IEEE International Workshop on Quality of Service*, 2004.
[19] A. Kamra, V. Misra, and E.M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," *IEEE International Workshop on Quality of Service*, June, 2004.
[20] IBM Application Workload Manager, http://www.ibm.com/servers/eserver/xseries/systems_management/director_4/awm.html
[21] IBM z/OS Workload Manager, http://www-1.ibm.com/servers/eserver/zseries/zos/wlm/
[22] L. Ljung, *System Identification: Theory for the User (2nd Edition)*, Prentice Hall Information and System Sciences Series, Prentice Hall, 1999.
[23] C. Lu, T.F. Abdelzaher, J. Stankovic, and S. Son, "A feedback control approach for guaranteeing relative delays in Web servers," *IEEE Real-Time Technology and Applications Symposium*, 2001.
[24] C. Lu, X. Wang, X. Koutsoukos, "End-to-End utilization control in distributed real-time systems," *24th International Conference on Distributed Computing Systems*, March, 2004.
[25] Y. Lu, C. Lu, T. Abdelzaher, and G. Tao, "An adaptive control framework for QoS guarantees and its application to differentiated caching services," *IEEE International Workshop on Quality of Service*, May 2002.
[26] Y. Lu, A. Saxena, and T.F. Abdelzaher, "Differentiated caching services: A control-theoretical approach," *International Conference on Distributed Computing Systems*, 2001.
[27] P. Martí, R. Villa, J.M. Fuertes and G. Fohler, "On real-time control tasks schedulability", *European Control Conference*, 2001.
[28] Matlab System Identification Toolbox, http://www.mathworks.com/products/sysid/
[29] C. McCarthy, M. Murphy, and I. Subramanian, "Meeting performance goals with the HP-UX Workload Manager," *First Workshop on Industrial Experiences with Systems Software*, October, 2000.
[30] C.W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating system support for multimedia applications," *International Conference on Multimedia Computing and Systems*, 1994.
[31] Microsoft Virtual Server, http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx
[32] D. Mosberger and T. Jin, "Httperf --- A tool for measuring Web server performance," *Performance Evaluation Journal*, Vol. 26, No. 3, pp. 31-37, December 1998.
[33] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A resource-centric approach to real-time and multimedia systems," *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.
[34] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak, "Statistical service assurances for applications in utility grid environments," *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, October 2002.
[35] SUN Solaris Resource Manager, http://www.sun.com/software/resourcemgr/index.html
[36] VMware, www.vmware.com