Adaptive Entitlement Control of Resource Containers on Shared Servers

X. Liu
Univ. of Illinois at Urbana Chamapaign
201 N. Goodwin Ave
Urbana, IL, USA
xueliu@cs.uiuc.edu

X. Zhu, S. Singhal, M. Arlitt
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA, USA
{xiaoyun.zhu,sharad.singhal,
martin.arlitt}@hp.com

Abstract

In this paper, we describe the design of online feedback control algorithms to dynamically adjust entitlement values for a resource container on a server shared by multiple applications. The goal is to determine the minimum level of entitlement for the container such that its hosted application achieves desired performance levels. Classic control theory is used for both model identification and controller design. Specific implementation issues that affect the closed-loop system performance are discussed A self-tuning adaptive controller is also presented to handle limited variations in the workload. The controllers were implemented and evaluated on a testbed using the HP-UX PRM as the resource container and the Apache Web server as the hosted application in the container. In all experiments, our controller was able to quickly converge to the proper level of CPU entitlement for the Web server 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

Server consolidation is important for reducing infrastructure and management costs and increasing return on IT investment in enterprise data centers. Because applications sharing the same server may compete for system resources such as CPU, memory, and disk bandwidth, performance degradation and service level agreement (SLA) violations can occur under overload conditions. Two classes of technologies exist on the market today to provide isolation between co-hosted applications: server partitioning and server virtualization. For instance, HP's Process Resource Manager (PRM) [9] and IBM's Application Workload Manager [15] can partition a shared server to ensure each partition's entitlement to system resources under overload conditions. Similarly, server virtualization technologies [22][25] allow multiple virtual servers to be created on the same physical machine and encapsulate 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 provide performance isolation and service differentiation for applications on shared servers. However, current practices typically rely on offline capacity planning tools to statically determine the resource entitlement of each container before production, which does not fully utilize the benefit of statistical multiplexing between applications' resource demands. This is especially important for enterprise applications where static entitlement is difficult because resource demands vary over time due to changes in workloads. For resource containers that host such enterprise applications, we need to guarantee that the container always has enough resources to meet the performance goals of the hosted application.. 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 exposed APIs or configuration utilities provided by the container. The performance data may be provided to the controller by the application, or through a proxy agent that computes performance metrics from monitoring data.

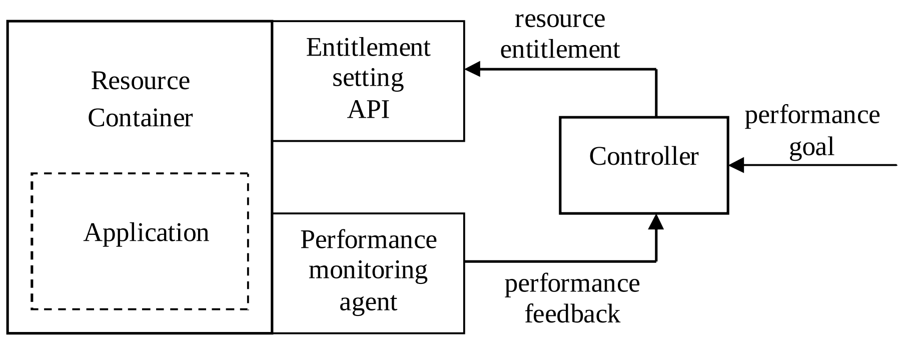


Figure 1. Entitlement control for a resource container using feedback

Most existing technologies for realizing this feedback loop online rely on policies or heuristics [10][15]. They require expert knowledge, and 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 using control theory as the foundation for designing the feedback loop in Figure 1. This provides useful guidelines for managing tradeoffs between system stability and performance. The system identification process utilizes a black-box approach to infer models from measurement data. The models are updated online in an adaptive control system. This is especially useful in environments where system operating conditions cannot be predicted accurately in advance. By using short-term dynamic models to predict system behavior, our approach can quickly react to changes in application requirements or system conditions, in complement to offline planning or 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 setup of our testbed. Section 4 demonstrates how a dynamic model can be inferred from experimental data using standard system identification techniques. 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 [4] was first proposed as a new operating system abstraction which separates a protection domain from a resource principal, thus enabling fine-grained resource management in servers. Our notion of resource containers is broader in the sense that it is agnostic of the specific technology used to create the container and how resource entitlement is enforced. In addition, the implementation of the resource container in [4] requires modification of both the kernel and the server applications, while our entitlement control system exploits externally exposed APIs or simple configuration utilities for the containers, and does not require a change in the applications running inside the containers.

The Rialto operating system [12] and Resource Kernels [23] provide operating system support for resource reservation, monitoring, and enforcement on shared servers. These resource reservation frameworks require that the resource entitlement for each application 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 typically leads 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.

Server partitioning or virtualization technology vendors also provide workload management tools for controlling application performance using feedback. Both the IBM z/OS workload manager [16] and the HP-UX Workload Manager [10] let the users define performance goals and priorities for applications, and automatically determine the amount of CPU and/or storage required for each application to meet the goals. The latter implements a proportional (P) controller that contains a number of parameters that have to be tuned by the users to achieve optimal performance of the controller. In this paper, we use control theory for designing such feedback control algorithms, thus minimizing hand-tuning required by the user.

Control theory has also been successfully applied to controlling performance or quality of service (QoS) for a variety of computer systems and software. Chapter 1 in [11] gives an extensive summary of related work in this area. The systems being controlled include Lotus Notes email server [7][8], Apache Web server [1][5][6][19], Squid proxy server [20], Lustre file system [13], as well as a 3-tier e-commerce site [14]. The output metrics include system-level metrics, such as CPU and memory utilization [1][6], cache hit ratio [20], and server queue length [5][8], application level metrics such as response time and throughput [13][14][19], or business level metrics such as profits [7]. Control mechanisms used include admission control or request throttling [5][13][14], Web content adaptation [1], application parameter tuning [6][7], resource allocation [19][20], and middleware [17][24]. The types of control algorithms used include variations of proportional, integral, and derivative (PID) control [1][5][8][17], pole placement [6], linear quadratic regulator (LQR) [6], fuzzy control [7][17], model predictive control (MPC) [1], and adaptive control [13][14][20]. Adaptive control, in particular, has received much interest due to its self-tuning capability that allows the controller to adapt to changes in operating conditions and workloads automatically.

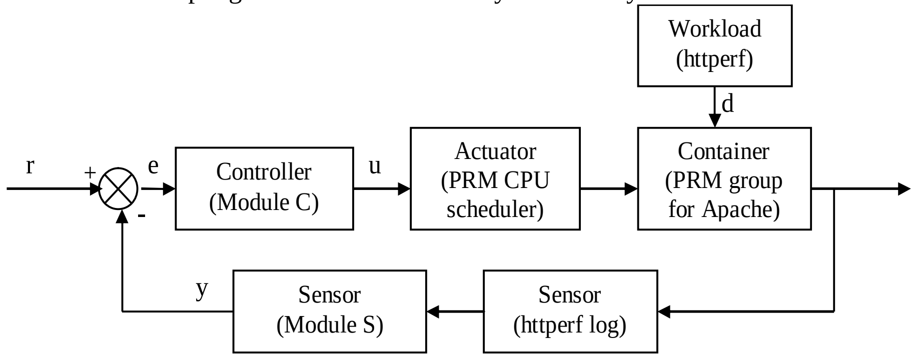
Our work is distinguished from prior work in that we propose a generic, non-intrusive approach for dynamically controlling resource entitlements that relies upon existing server partitioning or virtualization technologies and application capabilities. Our approach can be applied to any resource container and any application hosted inside the container 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 on a real system testbed.

3. Control System Architecture and Testbed Setup

In this section, we introduce a case study that uses the HP-UX PRM [9] 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 Resource container and actuator

In our case study, we use the HP-UX PRM as the resource container technology. 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.



$r(k) = RT_{ref}^{k}$ $y(k) = MRT^{k}$ $u(k) = CEP^{k}$ $e(k) = RT_{ref}^{k} - MRT^{k}$

Figure 2. Architecture of resource entitlement control system

In this paper, we focus on CPU as the key system resource for applications. We rely on the CPU scheduler in PRM as the actuator for our control system to enforce the CPU entitlement value 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.

3.2 Workload generator and sensor

We use *httperf* (ftp://ftp.hpl.hp.com/pub/httperf), a scalable client workload generator, to continuously send HTTP requests to the Apache Web server. We chose client-perceived mean response time, MRT (y), 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 completed during each sampling interval. The sampled MRT is fed into the controller as an input. Because the workload mix and intensity affect the degree of how system resources are stressed by the application, the workload also serves as a disturbance (d) to the control system.

3.3 Controller

The goal of the controller is to compute the proper level of CPU entitlement for the Web server in order to maintain the measured MRT around the desired response time. The latter is referred to as the 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 a controller module C. The controller module computes a new CPU entitlement value, CEP^{k} (u(k)), and feeds it into the actuator, PRM's CPU scheduler, which reallocates the CPU cycles during the current sampling interval.

3.4 Testbed setup

All experiments including system identification and control were conducted on a testbed of two computers connected by 100 Mbps Ethernet. 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 runs 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.

- **Server application:** We chose the Apache Web server release 2.0.48 as the application to be hosted inside a resource container. By implementing a thread model, the Apache 2.x releases are able to serve a large number of requests with less system resources than the previous process-based server in the Apache 1.3.x releases, yet still retaining the stability of a process-based server by keeping multiple processes available, each with a variable number of threads ready to serve incoming requests. During all the experiments, we use the default Apache server settings and configuration parameters recommended by the Apache group.
- **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 being a bottleneck. In addition, the total size of each working file set is small enough to fit in the file system buffer cache so that minimum disk activity was involved. 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 such 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.
- **PRM:** We use the version of PRM inside the HP-UX B11.00 kernel. 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. The CPU entitlement for the WEBSV PRM group is modified every time a new CEP is calculated. We enable CPU capping so that CEP also acts as an upper bound on the percentage of CPU cycles used by the WEBSV group. In addition, the real CPU utilization by the WEBSV group is measured every second and recorded in a log file to be analyzed after each experiment.
- **Sensor and controller:** Both the sensor module S and the controller module C are implemented in Perl. For the ease of parsing the httperf log, we placed both of 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). Complex behavior of the Web server makes it difficult to obtain such models using first-principles. We treat it as a black-box and infer the model from externally observable metrics using standard system identification techniques [17].

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 the MRT is roughly inversely proportional to the Web server's throughput, while the latter is proportional to the CPU entitlement. Therefore, the MRT is roughly inversely proportional to the CEP. Using the inverse of MRT allowed us to find a simple linear model as the basis for the controller design, while no such models could be found if the MRT were the output.

For our system identification experiments, we used the fixed workload WL1 at a rate between 300 and 600 requests/second to stress the server's CPU 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 CEP settings for both system identification and control. At the beginning of each sampling interval *k*, u(k) was randomly chosen and fed into PRM, and 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* [21], 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 [17] provides reasonably good prediction for the inverse of MRT for all the request rates:

$$y(k+1) = b_0 u(k) + a_1 y(k) \tag{1}$$

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

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

Notice that the system has a one-sample delay in the response of y(k) from u(k). Four potential factors contribute to this delay: first, the Web server itself may have a delayed change in the response time when its CPU allocation changes; second, y(k) is an average measure, which incurs up to one-sample 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.

Models with different parameter values were applied to the validation data, and the model-predicted output was compared to the measured output (inverse of MRT). One such comparison is shown in Figure 3, for data collected at the rate of 600 requests/second. The parameter values we chose are: b_0=0.17, and a_1=0.46. Among all the models we evaluated, this model has the highest r^2 value of 50.3%, where r^2 denotes the percentage of variation in the measured output accounted for by the model. Although r^2 is not the only measure for picking the best model, we notice that this model captures most of the fluctuations in the data with reasonable accuracy, while missing some of the peaks due to simplicity of the model. The results for other request rates are similar, with different values for b_0 and a_1.

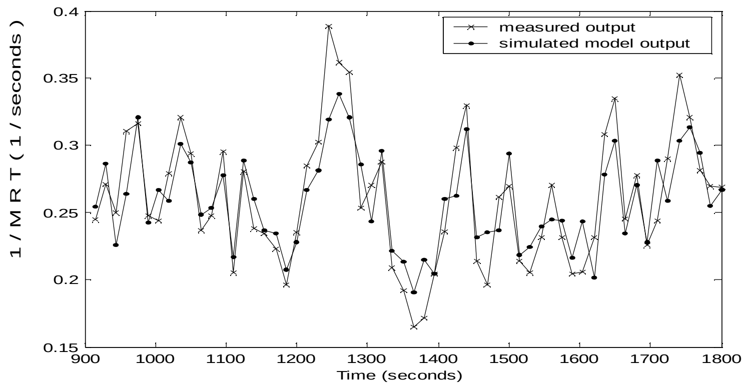


Figure 3. Comparison of measured output vs. model prediction

5. Offline Controller Design and Implementation

We now apply control theory to design the CPU entitlement controller in the feedback loop. The goal of the controller is to maintain the MRT around a given target by dynamically adjusting the CEP at every sampling interval. This is referred to as a regulation problem in the control literature.

The design criteria for choosing the controller algorithm and parameters are driven by the desired properties of the closed-loop system. First, the closed-loop system should be stable. More specifically, we want to minimize oscillation in the controlled metric, MRT; in control theory, this requires that the poles of the closed-loop transfer function be within the unit-circle. Second, the measured MRT should converge to the given response time target; in control theory, this is referred to as zero steady-state error. Finally, when the response time target 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.

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

$$u(k) = u(k-1) + (K_p + K_i)e(k) - K_p e(k-1), \tag{3}$$

where *e(k)=r(k)-y(k)* is the error between the reference value and the measured output. For our system, we have $e(k) = 1/RT_{ref}^k - 1/MRT^k$. Next we describe how the controller is implemented in the feedback loop.

5.1. Controller implementation design

We experimented with three controller implementation designs in our testbed. Due to space limitation, we only present one here. Interested readers are referred to [18] for detailed discussions and results of all three implementation choices. In all the experiments, the control interval is set to equal to the sampling interval, 15 seconds.

Figure 4 illustrates the timeline of our controller execution. We use a real-time timer to kick off the controller every 15 seconds. Starting from the beginning of each control interval T_k, instead of parsing the httperf log (by sensor S) immediately, a nap module (denoted by block N) is first executed. An immediate parsing would miss response time samples due to the buffered stream I/O in httperf for logging response times of completed requests. This may cause performance degradation of the closed-loop system, which we observed in our experiments [18]. By using the nap module, the controller waits for httperf to finish logging the response times of all the requests completed in the previous interval $[T_{k-1}, T_k]$. Note that simply turning off the streamed I/O in httperf may greatly impact its performance due to the large volume of requests seen by a typical Web server. We could also modify the httperf source code so that the logging is synchronized with our sampling interval. However, this becomes an intrusive approach and is not generic enough to be applied to all applications.

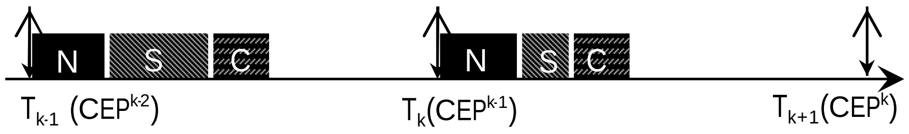


Figure 4. Controller execution timeline

The time to nap is a design parameter that should be chosen carefully such that a complete sample can be obtained, and at the same time, the whole sequence of control actions can be completed by the deadline (i.e. beginning of the next interval, T_{k+1}). In our experiments, we found that a nap time of 3 seconds was a good choice, as all requests were logged and no deadlines were missed.

The sensor module S follows the nap module N to do the parsing and outputs MRT^k. The controller module C then takes MRT^k as input along with RT_{ref}^k for computing control input CEP^k. To reduce control jitter, we need to place the control actuation time as close to the timer expiration time as possible. We achieve this by withholding the control input CEP^k calculated during the interval $[T_k, T_{k+1})$ until the beginning of the next interval, T_{k+1}, as illustrated in Figure 4. So CEP^k is communicated to the server at time T_{k+1}, and will be subsequently actuated by PRM.

However, with the implementation proposed above, the controller algorithm in equation (3) needs to be revised to reflect the extra delay caused by holding the control input till the next interval. This results in the following controller algorithm:

$$u(k+1) = u(k) + (K_p + K_i)e(k) - K_p e(k-1) . \qquad (4)$$

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:

$$C(z) = \frac{U(z)}{E(z)} = 1.75 \times \frac{z-0.46}{z(z-1)} . \qquad (5)$$

Analysis shows that the closed-loop system is stable, has zero steady-state error, and has rise time $t_r=45$ seconds (3 samples) and settling time $t_s=90$ seconds (6 samples).

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, these 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, 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.

We implemented a simple indirect self-tuning regulator [3] on our entitlement control testbed. 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 sampling 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 5 shows a block diagram for the self-tuning regulator, which is a variation of a general diagram in [3], but contains specific information on our system.

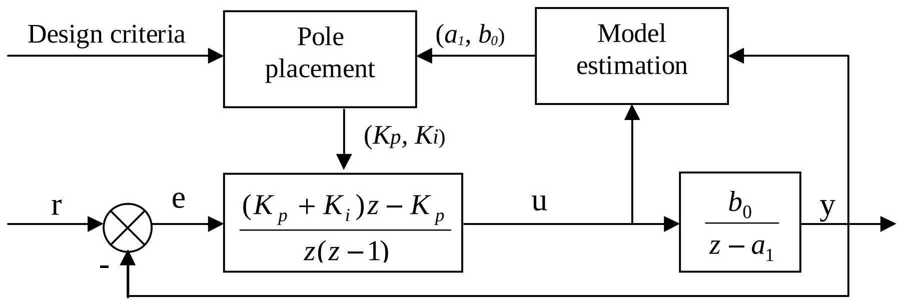


Figure 5. Block diagram of an indirect self-tuning regulator

We use the recursive least-squares (RLS) method [3] to estimate the two parameters for the AR model. The parameter values are updated in every control interval and are fed into a pole placement module. The latter chooses appropriate values for K_p and K_i in the PI controller such that the closed-loop system poles are at desired locations. The calculated gain values are then fed into the controller module for execution.

7. Performance Evaluation

In this section, we present the experimental results for the two controllers we described earlier in terms of the closed-loop system performance.

First, we used the fixed workload WL1 with a rate varying between 300 and 600 requests/second as what we used for system identification, to test the PI controller designed offline and its implementation. To avoid the starting phase of httperf, the controller was turned on at 60 seconds after httperf started, and the test lasted 30 minutes (120 sampling intervals). To test the system's response, we set RT_{ref} at 3, 5, 3 seconds during time intervals [0s-600s), [600s-1200s), [1200s-1800s), respectively.

Figure 6 shows the closed-loop system performance for workload WL1 at rate=600 requests/second using the controller in (5). Figure 6(b) shows the measured MRT and the RT_{ref} as a function of time. With a delay of 3-8 intervals, the controller could track changes in RT_{ref} and maintain the MRT within $\pm 15\%$ of the response time target.

Figure 6(a) shows the control input CEP^k and the measured CPU utilization of the WEBSV group during the experiment. For each RT_{ref} setting, our controller was able to quickly determine 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 at around 40-50%. The feedback controller carried out these adjustments automatically. 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 system model since the input data we used for system identification was CEP^k, not the real CPU utilization.)

Figure 6(c) shows the mean server throughput in each control interval in the number of completed requests per second. Since the mean throughput is inversely proportional to MRT, the control system regulates throughput the same way it does response times. 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 throughput of about 300 requests/second.

Next, we tested the self-tuning adaptive controller against the variable workload WL2. Figure 7 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.

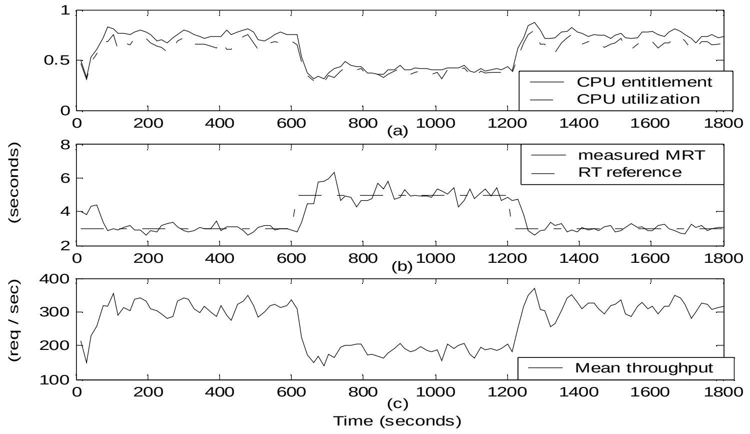


Figure 6. Closed-loop system performance of the fixed PI controller

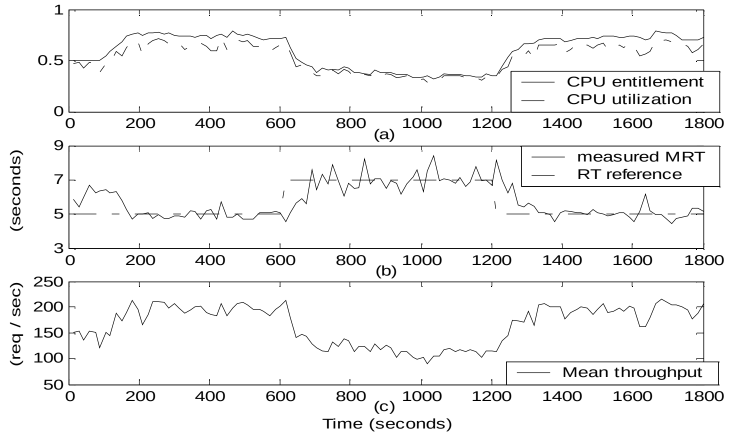


Figure 7. Closed-loop system performance of the adaptive controller

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.

8. Conclusions and Future Work

In this paper, we described the problem of designing online feedback control algorithms to dynamically adjust entitlement values for 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. 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 inclusion of throughput as another measured output would result in a multiple-input-multiple-output (MIMO) model for the controlled system. We intend to explore a broader set of techniques for designing feedback controllers for such systems.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their comments and suggestions that helped improve the quality of this paper.

References

[1] T.F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for Web server end-systems: A control-theoretical approach," *IEEE Trans. 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*, Prentice Hall, 1994.

[4] 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.

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

[6] 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.

[7] 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.

[8] S. Parekh, N Gandhi, JL Hellerstein, D Tilbury, TS Jayram, J Bigus, "Using control theory to achieve service level objectives in performance management," *Real Time Systems Journal*, vol. 23, no. 1-2, 2002.

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

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

[11] J.L. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*, Wiley-Interscience, 2004.

[12] 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.

[13] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance isolation and differentiation for storage systems," *12th IEEE International Workshop on Quality of Service*, June, 2004.

[14] A. Kamra, V. Misra, and E.M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," *12th IEEE International Workshop on Quality of Service*, June, 2004.

[15] IBM Application Workload Manager, http://www.ibm.com/servers/eserver/xseries/systems_management/director_4/awm.html

[16] IBM z/OS Workload Manager, http://www-1.ibm.com/servers/eserver/zseries/zos/wlm/

[17] B. Li and K. Nahrstedt, "A control-based middleware framework for quality-of-service adaptations," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1632-1650, 1999.

[18] X. Liu, X. Zhu, S. Singhal and M. Arlitt, "Adaptive entitlement control of resource containers on shared servers", HP Labs Technical Report, available at: http://www.hpl.hp.com/techreports/2004/HPL-2004-178.pdf.

[19] 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.

[20] Y. Lu, C. Lu, T. Abdelzaher, and G. Tao, "An adaptive control framework for QoS guarantees and its application to differentiated caching services," 10th *IEEE International Workshop on Quality of Service*, May 2002.

[21] System Identification Toolbox, http://www.mathworks.com/products/sysid/.

[22] Microsoft Virtual Server, http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx.

[23] 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.

[24] R. Zhang, C. Lu, T. Abdelzaher, J. Stankovic, "ControlWare: A middleware architecture for feedback control of software performance," *International Conference on Distributed Computing Systems*, Vienna, Austria, July, 2002.

[25] VMware, http://www.vmware.com.