Yaksha: A Self-Tuning Controller for Managing the Performance of 3-Tiered Web sites

Abhinav Kamra
Department of Computer Science
Columbia University
New York, NY 10027
Email: kamra@cs.columbia.edu

Vishal Misra
Department of Computer Science
Columbia University
New York, NY 10027
Email: misra@cs.columbia.edu

Erich M. Nahum
T. J. Watson Research Center
IBM Corporation
Yorktown Heights, NY 10532
Email: nahum@watson.ibm.com

Abstract— **Managing the performance of multiple-tiered Web sites under high client loads is a critical problem with the advent of dynamic content and database-driven servers on the Internet. This paper presents a control-theoretic approach for admission control in multi-tiered Web sites that both prevents overload and enforces absolute client response times, while still maintaining high throughput under load. We use classical control theoretic techniques to design a Proportional Integral (PI) controller for admission control of client HTTP requests. In addition, we present a processor-sharing model that is used to make the controller self-tuning, so that no parameter setting is required beyond a target response time.**

Our controller is implemented as a proxy, called *Yaksha*, which operates by taking simple external measurements of the client response times. Our design is non-invasive and requires minimal operator intervention. We evaluate our techniques experimentally using a 3-tiered dynamic content Web site as a testbed. Using the industry standard TPC-W client workload generator, we study the performance of the PI admission controller with extensive experiments. We show that the controller effectively bounds the response times of requests for dynamic content while still maintaining high throughput levels, even when the client request rate is many times that of the server's maximum processing rate. We demonstrate the effectiveness of our self-tuning mechanism, showing that it responds and adapts smoothly to changes in the workload.

I. Introduction

E-Commerce is rapidly becoming an everyday activity as consumers gain familiarity with shopping on the Internet [1]. Online merchants desire to maintain a continuous, consistent presence on the Web in order to keep customers satisfied and maximize both revenues and returns on their hardware and software investments. The infrastructure behind such E-Commerce Web sites is typically composed of a three-tiered architecture, consisting of a front-end Web server, an application server and a back-end database.

Two problems are frequently encountered with deploying such Web sites. First is *overload*, where the volume of requests for transactions at a site exceeds the site's capacity for serving them and renders the site unusable, frequently referred to as the "slashdot effect." Named after the Web site slashdot.org, this occurs when a huge user base is referred to a previously undiscovered Web site, which in turn is overwhelmed by the sudden volume and crashes. Second is *responsiveness*, where the lack of adequate response time leads to lowered usage of a site, and subsequently, reduced revenues. This is sometimes called the "abandoned shopping cart" problem. Both issues are instances of a larger problem: given the unpredictability of Web accesses, how can a e-Commerce site provide responsive service to clients, even when user demand far outstrips the capacity of the site?

This paper presents a method for controlling multiple-tiered Web site performance, both by bounding response times and preventing overload. Our approach uses a *self-tuning proportional integral (PI) controller* for admission control, enabling overload protection and bounding response time based on an administrator-based policy (e.g., 90 percent of the requests should see a response time of less than 100 milliseconds). By using a self-tuning controller, our system automatically adapts to variation in load and requires only two parameter settings.

Our approach is distinguished through several features:

- *self-tuning*: By utilizing a self-tuning controller, our approach does not require parameterization of controller weights. The only input required is a desired response time. The remainder of the discovery process is automatic and the system can simply be placed in front of the site.
- *multiple tiers*: Our proposal works with complete Web sites consisting of multiple tiers, not just the Web server.
- *transparent*: Our method works with existing N-tiered Web sites, requiring no intervention or modifications to the Web server, application server, or database.
- *architecture-independent*: Related to the transparency issue above, many approaches are tied to the process-based server model used by Apache 1.3. Our approach works with any server model: process-based, thread-based, event-based, and even kernel-based.

Other proposals have needed extensive modifications to the operating system or a complete re-write of the server. Our method requires no changes to the operating system, Web server, application server or database. This allows rapid deployment and use of pre-existing components.

We present an implementation of our controller in a proxy, called *Yaksha*[1]. We evaluate our system with standard software components used in multiple-tiered e-Commerce Web sites, namely Linux, Apache, Tomcat, and MySQL. We drive the

[1]*Yakshas* are the guardians of wealth in the Hindu pantheon.

system using the industry-standard TPC-W benchmark, and demonstrate that Yaksha achieves both stable behavior during overload and bounded response times. Our results show that a properly designed and implemented controller be used in a complex environment, such as multiply-tiered Web sites.

The remainder of this paper is organized as follows: Section II overviews related work and contrasts our approach to previous proposals. Section III provides the formal description for our controller. Section IV describes our controller implementation, experimental testbed, and methodology. Section V shows our experimental results in detail. Finally, Section VI summarizes our conclusions and offers possible directions for future work.

II. Background

In this section, we first provide a survey of previous work on admission control and then contrast those works with our approach.

A. Previous Work

Much related work has been done in the areas of overload control, admission control, service differentiation and quality of service (QoS) for Web servers. Due to space limitations, we provide a very brief overview here.

Priority and Service Differentiation: Early works focused simply on differentiating service to different classes of customers based on priority [2], [3]. Here the idea was simply to provide *better* service to higher-priority customers, without attempting to provide either relative or absolute service guarantees. Almeida et al. [2] compare user-level and kernel-level request scheduling policies, showing that high-priority requests can receive improved response times at the expense of lower-priority requests. Eggert and Heidemann [3] compare three application-level mechanisms for providing service differentiation. More recently, McWherter et al. [4] study this issue in the context of databases.

Kernel Mechanisms: Other research has investigated admission control, sometimes referred to as overload control. Mogul and Ramakrishnan [5] demonstrate how packet arrivals in software-based routers could create a scenario called "receive livelock," a form of overload. They show how a polling-based approach, instead of an interrupt-driven one, can eliminate this problem. Druschel and Banga [6] show a similar concept in the context of Web servers, demonstrating how a network subsystem architecture that isolates processing costs could provide improved stability and throughput under high loads. Voigt et al. [7] study different kernel and user-space mechanisms for admission control and service differentiation in overloaded Web servers. They evaluate their proposed mechanisms in AIX with a static content workload and find that the kernel-based mechanisms provide better performance.

Combining Admission Control with Differentiation: More recent approaches seek to combine differentiated service with admission control. Bhatti and Friedrich [8] proposed an architecture for Web servers to provide QoS to differentiated clients, incorporating request classification, admission control, and request scheduling. They provide an example implementation using Apache. While they show how premium clients receive preferential service over basic clients, they do not experimentally demonstrate sustained throughput in the presence of overload when *premium* requests outstrip capacity. In addition, their implementation and evaluation ignore dynamic content.

Li and Jamin [9] provide an algorithm for allocating differentiated bandwidth to clients in an admission-controlled Web server based on Apache. They evaluate their algorithm using Apache with static content. Bhoj et al. [10] present the Web2K mechanism, which prioritizes requests into two classes: premium and basic. Connection requests are sorted into two different request queues, and admission control is performed using two metrics: the accept queue length and measurement-based predictions of arrival and service rates from that class. The authors evaluate their system using Apache, and show how high priority requests maintain stable response times even in the presence of severe overload. They emulate the use of dynamic content, using a very simple model of execution costs where a CGI script runs up to 25 milliseconds.

Pradhan et al. [11] present an observation-based framework for "self-managing" Web servers that adapt to changing workloads while maintaining QoS requirements of different classes. Their evaluation is done primarily using static workloads, although they also examine a synthetic CGI script that blocks for a random amount of time. Kanodia and Knightly [12] propose a mechanism that integrates latency targets with admission control. Using both request and service statistical envelopes, the mechanism improves the percentage of requests that meet their QoS delay requirements. The authors evaluate their scheme via trace-driven simulation.

Control-Theoretic Approaches: Several researchers have examined how control theory can be applied in the context of Web servers. Lu et al. [13] present a control-theoretic approach to provide guaranteed *relative* delays between different service classes. Abdelzaher et al. [14] propose using classical control theory for Web servers to provide performance isolation (i.e., isolating different virtualized servers from each other), service differentiation, and QoS adaptation (similar to service degradation). They provide an implementation using the Apache Web server. Diao et al. [15], [16] advocate a similar approach, using control theory to maintain Apache's KeepAlive and MaxClient parameters, showing quick convergence and stability. However, these parameters do not directly address metrics of interest to the Web site, such as response time or throughput. The authors also limit their attention to static content. Lassettre et al. [17] apply control theory to the *provisioning* problem in the application server tier. They use a control-theoretic approach for dynamically swapping application servers in and out in order to meet a response time target. They do not address admission control, however, for when requests exceed available resources.

Admission Control for Dynamic Content: Welsh and Culler [18] describe an adaptive approach to overload control in the context of the SEDA [19] Web server. SEDA decomposes

Internet services into multiple stages, each one of which can perform admission control. By monitoring the response time through a stage, each stage can enforce a targeted 90th-percentile response time. Admission is controlled by using an additive-increase multiplicative decrease (AIMD) control algorithm that adapts the rate at which tokens are generated for a token bucket traffic shaper. The control parameters are set by hand after running tests against the system. Their evaluation includes dynamic content in the form of a web-based email service. Elnikety et al. [20] demonstrate a method for admission control in 3-tiered E-Commerce Web sites. Their approach requires an administrative setup phase in order to determine the capacity of the system, which is then fed back into the admission control mechanism.

Other Issues in Admission Control: Cherkasova and Phaal [21], [22] identify that users interact with Web sites in sessions comprised of multiple requests, and thus that performing admission control on requests in isolation can cause sessions to be aborted unnecessarily. They show that by considering session characteristics in admission control, rather than just individual requests, fewer sessions will be rejected. Chen et al. [23] propose a very similar concept. Schroeder and Harchol-Balter [24] show how preferential scheduling can be used to improve the response time of static content requests during *transient* overload conditions.

B. Contrasts to Our Approach

Our approach differs from the above works in many respects; we broadly outline them here based on several categories. The full description of our design is presented in Section III.

Self-Tuning: Unlike other control-theoretic approaches to admission control [14], [15], [16], [13] Yaksha utilizes a self-tuning proportional integral (PI) controller. The self-tuning and design is done based on the queuing model abstraction that we use in our paper. While approaches like system identification can lead to models that might be more accurate on a per-system basis, our approach provides a general and simple model and is not specific to the details of the system architecture, e.g., the particular web-server, application server or database server employed. This eliminates the requirement for off-line measurements and parameter setting that is present in these other approaches. By minimizing the learning aspects of the controller, we aim to design a system that is portable across different architectures of 3-tiered websites. Typically the assumption is made that service costs are linear in proportion to the size of the response generated. As will be seen, dynamic content service times have much greater variability which has no relation to the size of the response generated. Using a self-tuning controller makes our system more robust to variation and unpredictability in the workload.

Multiple Tiers and Dynamic Content: Most of the above work has only addressed static content, a much simpler workload, whereas our approach is fundamentally concerned with multiple-tiered Web sites that include dynamic content and back-end databases. Of the few that do considered dynamic content, two use a simple linear approximation of the service cost in the form of a dummy CGI script [10], [11]. We use a full implementation of the dynamic functionality and incur actual execution costs, which can vary by orders of magnitude.

Relatively few works are closely related to ours in terms of being implemented (as opposed to modeled or simulated) and addressing dynamic content: Neptune [25], Aron's resource management framework [26], SEDA [18], [19], and Gatekeeper [20]. Neptune and Aron's framework both use search as a dynamic workload, whereas we use a transaction-oriented e-Commerce workload. SEDA's evaluation includes dynamic content in the form of a custom email service driven by a home-grown workload generator. Our evaluation is performed using generic software components and driven using TPC-W, an industry-standard e-Commerce workload. While SEDA's approach is perhaps more general, Yaksha has the advantage that it is completely transparent to the Web site and is thus more easily deployable. In addition, SEDA currently requires a complex administrator-driven discovery process to determine controller weights. Gatekeeper similarly requires a configuration stage to determine the capacity of the system. Yaksha, on the other hand, requires much less parameterization. Instead, it is driven by a self-tuning controller and is thus more robust to variation in load. In addition, Gatekeeper has only been evaluated with a database, whereas our system protects all three tiers.

Transparency and Architecture-Independence: Virtually all the above approaches require invasive surgery to the affected systems. Many modify Apache [14], [10], [8], [27], [15], [16], [3], [9], [13], [11] or require a completely new architecture [25], [18]. Our approach preserves investment in pre-existing infrastructure, and allows for rapid deployment. In addition, Yaksha's architecture-independence allows use with servers utilizing any concurrency architecture. For example, many previous works are based on Apache 1.3, which is process based, yet Apache 2.0 is now thread-based for performance. IIS is also thread-based, and many sites use Tux [28], a kernel-based server, to handle static content. As a transparent reverse proxy, Yaksha can inter-operate with any of these architectures.

Other Contrasts: Finally, many previous works have attempted to identify overload through indirect measurements such as queue length or bandwidth utilization. Others have taken the direct indicator of response time, but considered only relative performance guarantees rather than absolute guarantees. Yaksha addresses the concept of load directly by dropping requests.

III. Modeling and Design

In this section we present the system model and design procedure for Yaksha. Our abstraction for the E-commerce server is an $M/GI/1$ Processor Sharing queue. This abstraction encapsulates everything that sits *behind* Yaksha.

We begin by introducing some notations. We denote by $T(x)$ the mean response time of a job whose job size (or service time) is x. The job size in an $M/GI/1$ is an i.i.d.

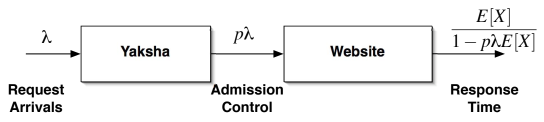

Fig. 1. System model for Yaksha and the Web site

.

random variable, denoted by X, whose probability distribution function is $F(x)$, with a mean $E[X]$.
It is well known that

$$T_{\text{PS}}(x) = \frac{x}{1-\rho}. \tag{1}$$

where ρ is the load of the queue. The mean response time for all jobs, T_{RT}, is simply

$$T_{RT} = \int_0^\infty T_{\text{PS}}(t)dF(t), = \frac{E[X]}{1-\rho} \tag{2}$$

The task of Yaksha is to control T_{RT}, by controlling an acceptance probability p_a of requests to the system. We model the feedback control system using a fluid model. Fluid models have been very successful in the analysis and controller design for network flows [29], [30]. Similar to their model, we work with the *mean* value of the response time as the variable to control. One important difference between their approach and our approach is that while they were controlling the mean queue length, that mean queue length has also been shown [31], [32] to be the deterministic fluid limit (as opposed to a mean valued fluid model) of the sample path as the number of flows is scaled. Thus, the queue and the controller in their model deals effectively with a sample path quantity directly. The difficulty in our case is that the mean value is not a fluid limit, and we don't have access to the mean value directly. Instead, we use smoothed, measured values of the response time as our estimate of the mean. This smoothing and averaging is assumed to be part of the system, and we do not have a separate low pass filter to model this aspect of the system.

Returning to the system model, it is depicted in Figure 1. Requests arrive to Yaksha with a mean rate $\lambda(t)$, and get modulated with an admission probability $p_a(t)$, with the Web server observing a mean arrival rate of $p_a(t) \cdot \lambda(t)$. The mean response time is then given by

$$T_{RT}(t) = \frac{E[X]}{1 - p_a(t)\lambda(t)E[X]}$$

Similar to previous approaches, we assume that there is an operating point p_0 of the system, that achieves $T_{RT} = T_{ref}$ for a given λ, where T_{ref} is our desired response time. We then linearize the response function about the operating point, and work with a small signal model. The linearization is a simple gain function, which is the derivative of the expression for the mean response time with respect to p_a, evaluated at p_0, i.e.,

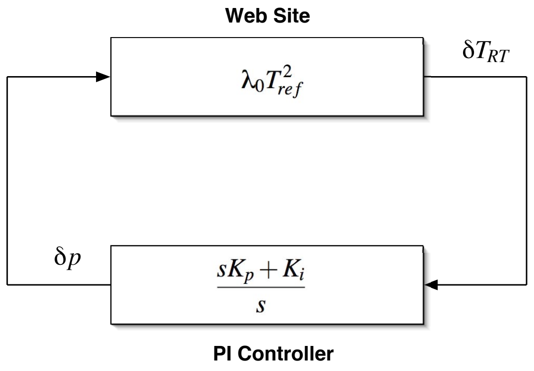

Fig. 2. Small signal model for the feedback loop

.

$$\begin{aligned}\frac{\partial T_{RT}}{\partial p_a} &= E[X]\frac{\lambda_0 E[X]}{(1-p_a\lambda E[X])^2}|_{p_a=p_0} \\ &= \lambda_0 T_{ref}^2\end{aligned}$$

This yields

$$\delta \dot{T_{RT}}(t) = \lambda_0 T_{ref}^2 \delta \dot{p_a}(t)$$

Where

$$\begin{aligned}\delta T_{RT} &\doteq T_{RT} - T_{ref}; \\ \delta p_a &\doteq p_a - p_0\end{aligned}$$

The linearized closed loop system model is then depicted in Figure 2. We control δT_{RT}, the deviation of T_{RT} from the equilibrium value, and our feedback signal is δp_a, again the deviation of p_a from the equilibrium value p_0. The feedback signal can be provided by any controller, our choice for this problems is the PI controller [33]. We use integral control since we desire zero steady state error in the response time. A pure proportional controller is unable to control the response time to a specific value. The PI controller has a transfer function given by

$$PI(s) = K_p + \frac{K_i}{s}$$

in the frequency domain. A PI controller has the desirable property that the steady state error of the controller is 0, in other words it can control the input to any desired value. Care must be taken however, to ensure that the controller is stable. We briefly outline our design process. The closed loop transfer function of the system, $C(s)$ is given by

$$C(s) = \frac{\lambda_0 T_{ref}^2 PI(s)}{1 + \lambda_0 T_{ref}^2 PI(s)}$$

We first assume a nominal value of $E[X]$, the mean job size of the workload distribution. As it will become clear in the self-tuning section, the exact value is not important. Using the relationship $\lambda E[X] = \rho$, given a value of $E[X]$, we can compute λ_0 from Equation (2). The design algorithm is simple: we first fix the time constant of the controller to 10 seconds, that fixes the closed loop pole at s = -0.1 (i.e., the denominator of the closed loop transfer function evaluates to 0 at $s = 0.1$). Next we provide a phase margin of 45, see [33] for detailed explanations, but in a nutshell phase margin reflects the relative stability of the controller: the amount of deviation the controller can tolerate in the system parameters before instability sets in. Note here instability refers to oscillatory behavior, rather than unbounded growth of any queue.

The two design constraints together fix the parameters of the base controller. We keep a large phase margin because we do not want our controller to be too aggressive. The response time samples that we obtain from the system are smoothed values, but still they are not true means. Hence we want the controller to operate at a slower time scale to allow the response time to converge to a mean. The design process yields a design in the continuous domain, however it must be implemented digitally. Our nominal loop transfer function is

$$L(s) = K_g(K_p + \frac{K_i}{s}) \tag{3}$$

where K_g is given by $K_g = \lambda_0 T_{ref}^2$

Choosing a phase-margin of -135 degrees gives us the following relation:

$$|K_p| = |\frac{K_i}{\omega}| \tag{4}$$

At the natural frequency of the system ω_g, using $|L(j\omega)| = 1$ gives us

$$K_g\sqrt{K_p^2 + \frac{K_i^2}{\omega_g^2}} = 1 \tag{5}$$

Using Equations 4 and 5 at $\omega = \omega_g$, we get the following equtions for the design parameters.

$$K_p = \frac{1}{\sqrt{2}K_g} \tag{6}$$

$$K_i = -POLE * (K_p + \frac{1}{K_g}) \tag{7}$$

where $POLE$ is the pole value we choose.

After using bilinear transform [34] to convert Equation 3 into digital form, we get the following equations for A and B.

$$A = K_p + \frac{K_i}{2f_T} \tag{8}$$

$$B = K_p - \frac{K_i}{2f_T} \tag{9}$$

and the controller executes the following update mechanism at fixed intervals:

$$p_d := A * (T - T_{ref}) - B * (T_{old} - T_{ref}) + p_{old}$$
$$p_{old} := p_d$$
$$T_{old} := T$$

Here p_d is the drop probability, which is simply $1-p_a$ where p_a is the admission probability. T is the smoothed averaged response time as measured at the proxy. A and B are the design parameters of the PI controller.

Next we outline the self-tuning aspects of the controller.

A. *Self-tuning of the PI controller*

The parameters of our controller are decided by the placement of the closed loop pole and our specification of the phase margin. Both of these quantities depend on the open loop transfer function of the plant. Earlier we had derived the open loop transfer function to be a pure gain value, the partial derivative of the response time function. We want the controller to be robust to variations in the workload distribution. Apriori we have no knowledge of the mix of servlets in the request distribution, the arrival rate of individual requests and indeed the operating point of the system. In other words, we do not know either $F(X)$ or the arrival λ, which makes the design task difficult. What we do know, is that $F(X)$ is a weighted sum of job sizes for the individual servlets. The weights are decided by the rate at which requests for individual servlets arrive. If the workload changes, λ_0 changes. We use this fact to estimate the system parameters at the server in the following manner. Recall that we have

$$K_g = \lambda_0 T_{ref}^2$$

where we assume a nominal λ_0 in the design process. However, the *effective* arrival rate of jobs that the system observes is $p_a\lambda$, where p_a is the admission probability that the controller computes. Hence, by observing p_a, assuming p_a converges, we can estimate the true arrival rate λ.

Now if the workload changes, the controller needs to self-tune. To accomplish this, we set $K_g = \frac{\lambda_0 T_{ref}^2}{p_a}$, where λ_0 is the nominal arrival rate we used in the design process. Then successively using Equations 6, 7, 8 9, we get the new values of A and B. The controller keeps a running average of p_a as

$$p_a = \alpha p_a + (1-\alpha)(1-p_d) \tag{10}$$

where $\alpha > 0$ is a constant that operates on a much slower time scale. The moving average and adaptation is done over 10 seconds to allow p_a to converge. With the technique outlined above, we come up with a lightweight, self-tuning controller.

IV. Implementation

In this section we describe the experimental testbed we use, the experimental methodology and the hardware and software environment.

A. *Controller Implementation*

We use tinyproxy v1.6.1 [35] and modify it to act as a controller. Tinyproxy is a lightweight HTTP proxy which is much faster and consumes fewer resources than regular HTTP proxies. All HTTP requests from the client are directed towards this proxy, which then relays them to the Web and application server after the control decision. All responses from the Web and application server to the client also go through the proxy. Figure 3 displays the baseline measurement of response times of the various servlets under no load.

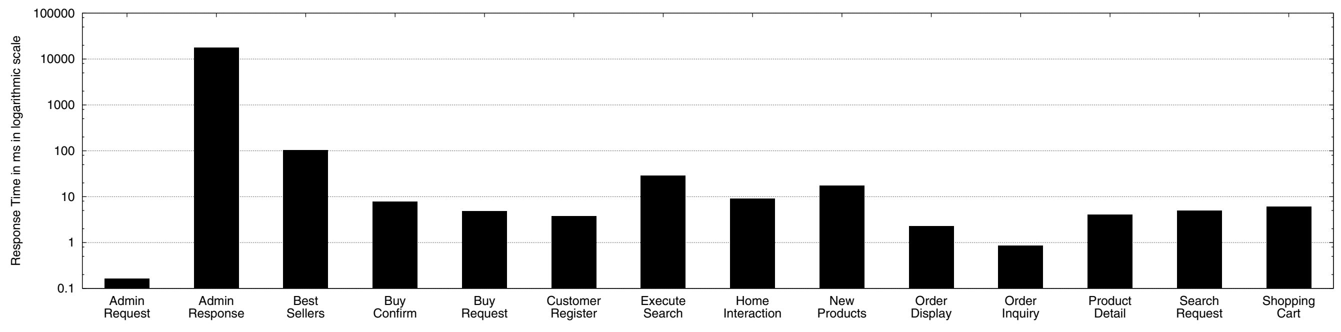

Fig. 3. TPC-W Servlet Service Times (no load)

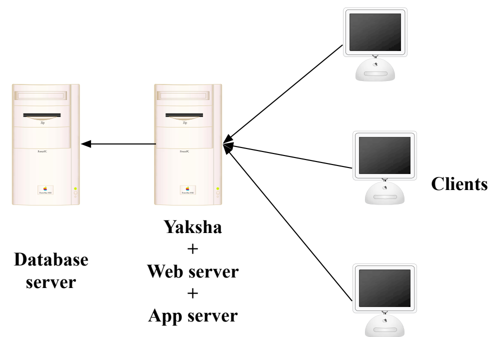

Fig. 4. Testbed Setup

We inserted a controller module in tinyproxy, which distinguishes different types of servlet queries. The controller also measures the response times of each of the HTTP requests which is then used as feedback for the control loop.

B. Hardware and Software

The overall structure of our testbed is shown in Figure 4. It consists of five machines. Three of them are used as clients to generate HTTP requests. One machine is used as the Web and application server and another hosts the database server. The client machines drive the system with a workload generator which is described in more detail in section IV-D below.

Each machine is a 2.8 GHz Intel Xeon, 1 GB RAM PC with Gigabit Ethernet connected point-to-point full duplex with the switch. All machines run RedHat Linux kernel 2.4.20. We use Jakarta Tomcat v4.1.27 [36] as the Web and application server and MySQL v4.1.0-max-alpha [37] as the database server.

The controller embedded inside tinyproxy resides on the machine hosting the Web and application server.

C. TPC-W Benchmark

When evaluating Web server performance, a *workload generator* is typically used to drive the system in a hopefully representative manner. We use what is effectively the current standard workload generator for e-Commerce sites, TPC-W [38], [39]. The TPC-W benchmark from the Transaction Processing Council (TPC) is a transactional Web benchmark specifically designed for evaluating e-commerce systems. It is meant to model a "typical" e-commerce site, in the form of an online bookstore. The TPC-W specification requires 14 different interactions, each of which must be invoked with a particular frequency. More detail about TPC-W can be found in [38]. The database contains multiple tables that are meant to represent the data needed to maintain a real site, including customers, addresses, orders, credit information, individual items, authors, and countries. We scaled the TPC-W database to 10,000 items and 288,000 customers, which corresponds to 350 MB of data.

TPC provides a specification but not source code. We thus use the freely available Java TPC-W implementation developed by the PHARM research group at the University of Wisconsin-Madison [40]. The implementation captures all the functionality required by the TPC-W specification that affects performance and is consistent with the official TPC-W specification version 1.0.1.

D. Client Workload Generator

The Java TPC-W implementation includes a workload generator, which is a standard closed-loop session-oriented client emulator. Each emulated browser represents a virtual user. The amount of load generated is determined by the number of emulated browsers. Each client opens a session to the front-end Web server using a persistent HTTP connection, issues a series of requests for the duration of the session, and then closes the connection. Within each session, the client repeatedly makes a request, parses the server's response, waits a variable amount of time, and then follows a link embedded in the response. The server's response is a Web page consisting of the answer to the queries in the request, and contains links to the possible set of pages that the client can transition to from this response. A finite-state Markov model is used to determine which subsequent link from the response should be followed, using a transition matrix with probabilities attached to each transition from one state to another. Each state in the transition matrix corresponds to a particular interaction defined in the TPC-W specification. The variable amount of time between requests is called the *think time*, and is intended to emulate a real client who takes some period of time before clicking on the next request. Think time is exponentially distributed with a mean of 0.7 seconds and bounded at a maximum of 7 seconds.

V. Experimental Results

In this section we present detailed experimental results to show the effectiveness of the PI controller in controlling response times while maintaining high levels of throughput.

A. Controller Overhead

The Yaksha controller is implemented as an HTTP proxy. All HTTP requests from the client are handled by this proxy which acts as a relay between the clients and the Web and application server. We show that adding this proxy component to the overall system has negligible overheads.

Figure 5 shows the variation of average response times with increasing load when clients connect directly to the Web and application server. The experiment is repeated but now with the proxy relaying all connections (i.e., no controller is used and no requests are dropped). The nearly identical response time curves show that adding the proxy mechanism does not have any significant impact on the system performance. Note that the response times we plot here are measured at the client, and they should not be directly compared to the response times measured at the proxy that we plot later on in this section. Response times at the client have additional overheads, including network delays and client processing delays, e.g., Java virtual machine overheads.

From now on, for the rest of the experiments we use the Yaksha controller embedded in tinyproxy for admission control and call this the "controlled" experiment. Furthermore for comparing with the situation when there is no admission controller, we slightly modify the same Yaksha controller so that it always accepts new connections. We use this modified controller as a "Null Controller" and call the experiments "uncontrolled". This allows us to measure response time and throughput from the same perspective, namely, at the proxy. Figure 6 is a validation of the processor sharing abstraction that we employ for our design. Recall that

$$T_{\text{PS}}(X) = \frac{E[X]}{1-\rho} \tag{11}$$

where ρ is $\frac{\lambda}{\mu}$, with λ being the job arrival rate and μ the capacity of our abstract processor (encompassing all the three tiers of the Web site). Now given a particular response time T_1, we have

$$T_1(X) = \frac{E[X]}{1-\frac{\lambda_1}{\mu}}$$

Assuming an average job size $E[X]$, we can compute the processor capacity if we know the response time and the job arrival rate. In Figure 6 we plot the empirically computed processor capacity, $\hat{\mu}$, as we slowly vary the number of emulated browsers using the system in an experiment. Both the response time and the number of requests (the empirical arrival rate λ) are smoothed over 10 seconds. The plot shows a fairly constant value of the empirical processor capacity, with a mean of about 50 jobs per second, thereby validating the processor sharing abstraction.

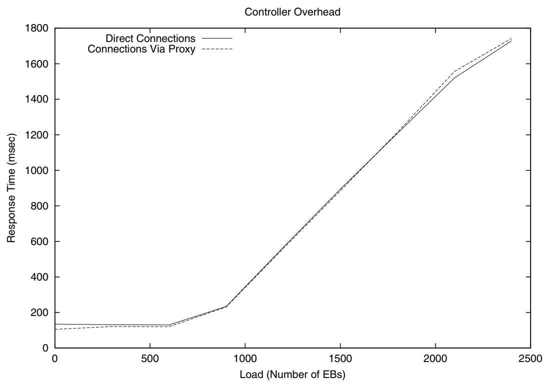

Fig. 5. Controller Overhead: Response Time With and Without Proxy

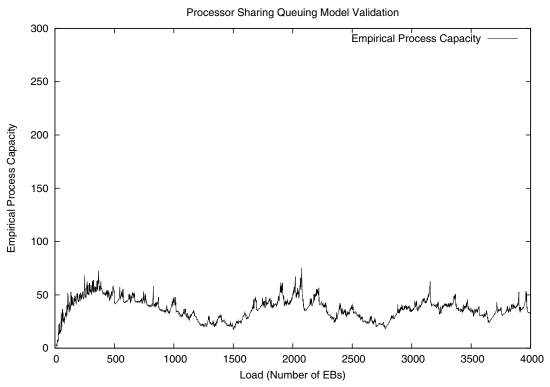

Fig. 6. PS Queuing Model Validation

B. Response Time Control at Overload

As client load increases, the throughput increases until the load reaches a threshold after which the throughput drops and response times grow. A major motivation for using a admission controller is to maintain reasonable throughput levels at high loads. We show that the Yaksha controller is able to bound response times and maintain significant throughput levels even at excessive load levels.

Figure 7 shows how throughput varies with increasing load for both controlled and uncontrolled experiments. As can be seen, Yaksha is able to maintain significant throughput levels even at very high load levels. Note the downward trend for the uncontrolled case as overload takes its toll.

Figure 8 shows the average response times of the servlets for increasing load for both the "controlled" and "uncontrolled" cases. As can be observed, Yaksha prevents server overload and so is able to effectively bound request response times while maintaining high throughputs. We set the reference value T_{ref} to be 150ms for the experiment, and that is the value Yaksha tries to control the running average of the response time. Figure 9 shows the cumulative throughput variation for the same experiment.

C. Adaptation to Changing Workload

Adapting effectively to changing client workloads is a necessity for a good admission controller. In the next set of

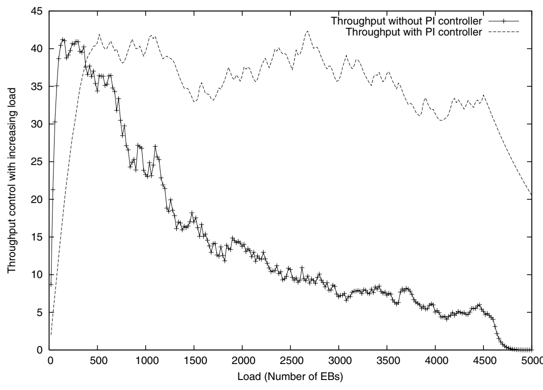

Fig. 7. Throughput control with increasing load

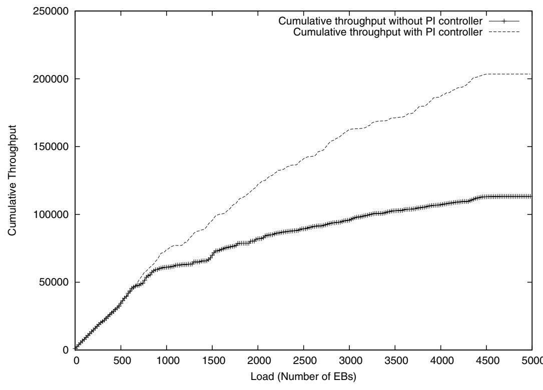

Fig. 9. Cumulative Throughput Control: Without PI control, throughput decreases sharply under heavy loads

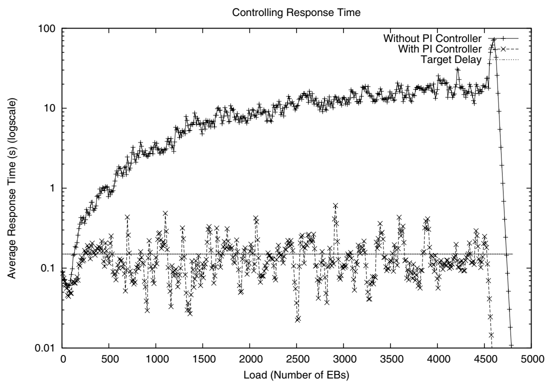

Fig. 8. Response Time Control: Without the PI controller the response time can grow unbounded

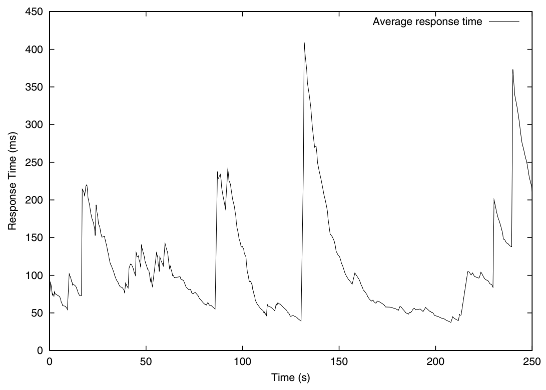

Fig. 10. Response Time variation under changing workloads

experiments we demonstrate the effectiveness of the Yaksha adaptation (self-tuning) process. Figures 10, 11,12 and 13 show how the various parameters of the controller adapt to changing workload scenario.

TPC-W has three types of in-built workload mixes, namely the “Browsing Mix”, the “Shopping Mix” and the “Ordering Mix”. The three generators have distinct frequencies of requesting different types of servlets and hence different workload characteristics.

To test how well Yaksha adapts to changing workloads, we conducted the following experiment, highlighted in Figures 10–13. In these experiments, we control for an average response time of 150 milliseconds. At time 0, 700 EBs of the “Browsing Mix” start requesting transactions at the site. At time 50, 700 EBs more of the “Shopping Mix” join in, and then another 700 EBs of the “Ordering Mix” at time 100. Hence, in this experiment, not only does the load level change significantly, but the distribution mix of the workload also undergoes a dramatic variation as the experiment progresses.

Figure 10 shows how response time varies in reaction to the changes in workloads. Figure 11 plots the request drop probability over the period, and Figures 12 and 13 show the adaptation of the A and B parameters. As can be observed, at every arrival of an overload (new workload mixes start generating transactions), A and B adapt quickly and the controller is able to bring down the response time below the desired value. Towards the end of the experiment the response time starts building again, as the EBs start regenerating requests, and the controller is able to react quickly. We contrast this behavior with a static controller to bring out the benefits of self-tuning. In Figure 14 we plot the response times for the two types of controllers. The static controller is designed for the base case of 700 EBs, and as we observe it cannot handle overloads well. It is slow to react, and the response time stays well above the target value of 150ms. In another experiment, we design a controller that is more “aggressive”, it is designed for a nominal load of 2000 EBs. This static controller on the other hand displays oscillatory behavior, and again the performance suffers in comparison to Yaksha. This can be observed in Figure 15. The aggressive controller has response times oscillating above and below the target value between the arrival of bursts. This can be more clearly seen in Figure 16, where we plot the drop probabilities of the two controllers. While Yaksha responds to the burst arrivals, the static controller over-reacts, resulting in oscillatory behavior.

VI. Summary and Conclusions

In this paper we have designed an admission controller for 3-tiered websites based on classical control theoretic ideas. A fundamental aspect of our design is our use of a good

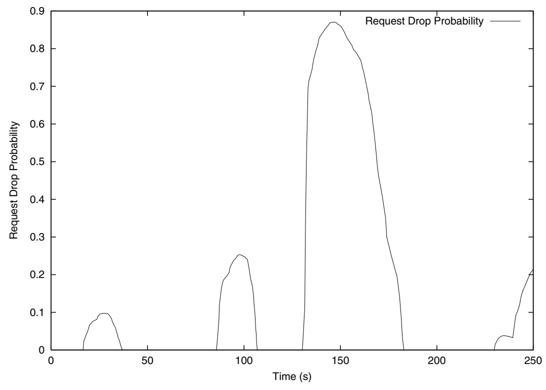

Fig. 11. Drop Probability variation under changing workloads

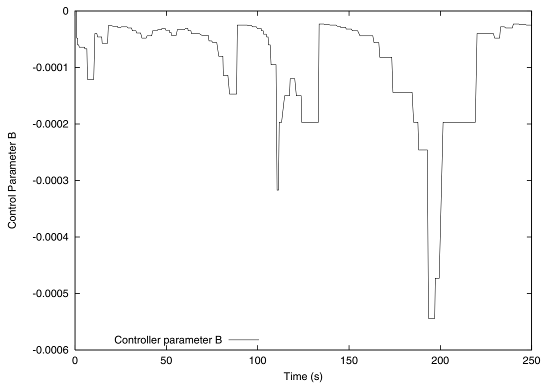

Fig. 13. B Parameter variation under changing workloads

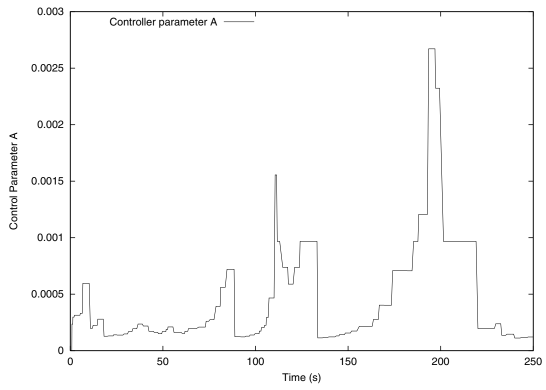

Fig. 12. A Parameter variation under changing workloads

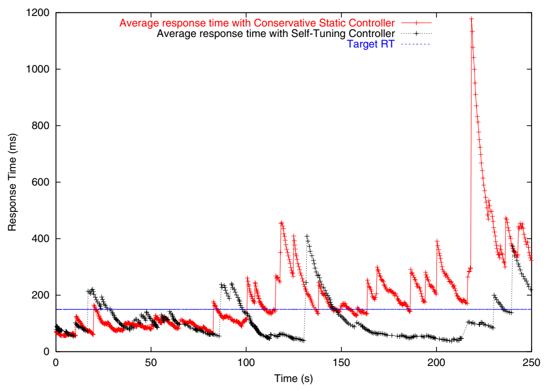

Fig. 14. Response time control: Static Conservative and Self-tuning controllers

modeling abstraction. With the help of our model, we are able to simplify the design process considerably, and the controller is completely self-tuning. We implement our controller design in the form of a proxy, Yaksha, and perform extensive experiments on a testbed using standard workload generators. Experimental results demonstrated that Yaksha is able to bound the response times for the requests and yet maintain a high throughput under overload. Moreover, Yaksha effortlessly adapts to varying loads and workload characteristics because of the underlying self-tuning design. A primary contribution of this paper is to demonstrate the benefits of using the right modeling abstraction for system design, in this specific case, an admission controller.

For future work, we are exploring further refinement of the model and the adaptation mechanism. Another avenue of further study is to add session awareness as opposed to request awareness. We are also analyzing real traces to come up with reasonable design parameters (e.g., time constants of the controller) for the system.

References

[1] News.Com, “E-commerce strong in third quarter,” November 2002, http://news.com.com/2100-1017-971123.html.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, “Providing differentiated levels of service in Web content hosting,” in *Workshop on Internet Server Performance*, Madison, WI, June 1998.

[3] L. Eggert and J. Heidemann, “Application-level differentiated services for Web servers,” *World-Wide Web Journal*, vol. 2, no. 3, pp. 133–142, August 1999. [Online]. Available: http://www.isi.edu/~johnh/PAPERS/Eggert99a.html

[4] D. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter, “Priority mechanisms for OLTP and transactional Web applications,” in *20th International Conference on Data Engineering (ICDE 2004)*, Boston, MA, March 2004.

[5] J. C. Mogul and K. K. Ramakrishnan, “Eliminating receive livelock in an interrupt-driven kernel,” *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997. [Online]. Available: citeseer.nj.nec.com/mogul96eliminating.html

[6] P. Druschel and G. Banga, “Lazy receiver processing (LRP): A network subsystem architecture for server systems,” in *Operating Systems Design and Implementation*, Seattle, WA, 1996, pp. 261–275. [Online]. Available: citeseer.nj.nec.com/druschel96lazy.html

[7] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra, “Kernel mechanisms for service differentiation in overloaded Web servers,” in *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[8] N. Bhatti and R. Friedrich, “Web server support for tiered services,” *IEEE Network*, vol. 13, no. 5, pp. 64–71, September 1999.

[9] K. Li and S. Jamin, “A measurement-based admission-controlled Web server,” in *IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[10] P. Bhoj, S. Ramanathan, and S. Singhal, “Web2K: Bringing QoS to Web servers,” HP Labs, Tech. Rep. HPL-2000-61, May 2000.

[11] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy, “An observation-based approach towards self-managing Web servers,” in *International Workshop on Quality of Service*, Miami Beach, FL, May 2002.

[12] V. Kanodia and E. W. Knightly, “Ensuring latency targets in multiclass Web servers,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 10, October 2002.

[13] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, “A feedback

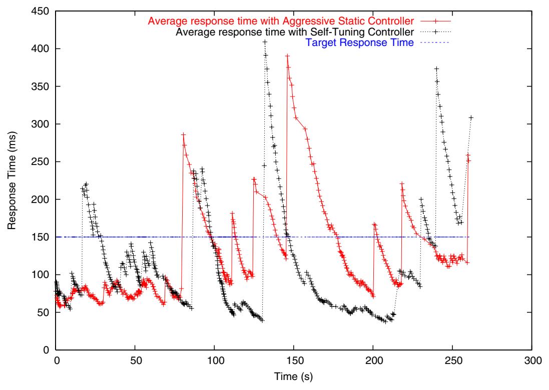

Fig. 15. Response time control: Static Aggresive and Self-tuning controllers

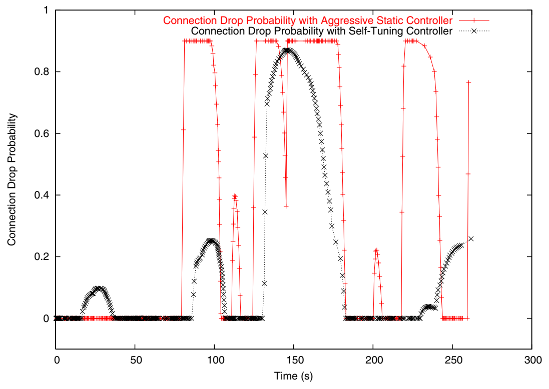

Fig. 16. Drop probability variation: Static Aggressive and Self-tuning controllers

control approach for guaranteeing relative delays in Web servers," in *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, June 2001.

[14] T. 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, no. 1, January 2002.

[15] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server," in *Proceedings of the Network Operations and Management Symposium*, Florence, Italy, April 2002.

[16] Y. Diao, X. Lui, S. Froehlich, J. L. Hellerstein, S. Parekh, and L. Sha, "On-line response time optimization of an Apache Web server," in *International Workshop on Quality of Service*, Monterey, CA, June 2003.

[17] E. Lassettre, D. Coleman, Y. Diao, S.Froehlich, J. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L., Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye, "Dynamic surge protection: An approach to handling unexpected workload surges with resource actions that have dead times," in *First Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, June 2003. [Online]. Available: citeseer.nj.nec.com/aron00cluster.html

[18] M. Welsh and D. Culler, "Adaptive overload control for busy Internet servers," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2003.

[19] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable Internet services," in *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

[20] S. Elnikety, E. M. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in dynamic e-commerce Web sites," in *International World-Wide Web Conference (WWW)*, New York, NY, May 2004.

[21] L. Cherkasova and P. Phaal, "Session-based admission control: A mechanism for peak load management of commercial Web sites," HP Labs, Tech. Rep. HPL-1998-119, June 1998.

[22] ——, "Session-based admission control: A mechanism for peak load management of commercial Web sites," *IEEE Transactions on Computers*, vol. 51, no. 6, June 2002.

[23] X. Chen, P. Mohapatra, and H. Chen, "An admission control scheme for predictable server response time for Web accesses," in *Proceedings of the 10th World Wide Web Conference*, Hong Kong, May 2001.

[24] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help," in *18th International Teletraffic Congress*, Berlin, Germany, August 2003.

[25] K. Shen, H. Tang, T. Yang, and L. Chu, "Integrated resource management for cluster-based Internet services," in *Operating Systems Design and Implementation*, Boston, MA, December 2002.

[26] M. Aron, S. Iyer, and P. Druschel, "A resource managment framework for predictable quality of service in Web servers," 2002, preprint vailable at http:/www.cs.rice.edu/~ssiyer/r/mbqos/mbqos-full.pdf.

[27] H. Chen and P. Mohapatra, "Session-based overload control in QoS-aware Web servers," in *IEEE Infocom*, New York, NY, June 2002.

[28] Red Hat Inc., "The Tux WWW server," http://people.redhat.com/~mingo/TUX-patches/.

[29] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong, "A control theoretic analysis of RED," in *IEEE Infocom*, 2001, pp. 1510–1519. [Online]. Available: citeseer.nj.nec.com/hollot01control.html

[30] ——, "On designing improved controllers for AQM routers supporting TCP flows," in *IEEE Infocom*, 2001, pp. 1726–1734. [Online]. Available: citeseer.nj.nec.com/hollot01designing.html

[31] F. Baccelli, D. R. McDonald, and J. Reynier, "A mean-field model for multiple TCP connections through a buffer implementing RED," in *Proceedings of PERFORMANCE*, Rome, Italy, August 2002.

[32] P. Tinnakornsrisuphap and A. M. Makowski, "Limit behavior of ECN/RED gateways under a large number of TCP flows," in *IEEE Infocom*, San Francisco, April 2003.

[33] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Addison-Wesley, 1995.

[34] K. J. Åström and B. Wittenmark, *Computer Controlled Systems: Theory and Design*, T. Kailath, Ed. Prentice-Hall, 1984.

[35] R. J. Kaes and S. Young, "tinyproxy," http://tinyproxy.sourceforge.net/.

[36] The Apache Jakarta Project, "Jakarta Tomcat servlet container," http://jakarta.apache.org/tomcat.

[37] MySQL AB, "The MySQL database," http://www.mysql.com.

[38] D. A. Menasce, "TPC-W: A benchmark for e-commerce," *IEEE Internet Computing*, May/June 2002.

[39] The Transaction Processing Council (TPC), "TPC-W," http://www.tpc.org/tpcw.

[40] T. Bezenek, H. Cain, R. Dickson, T. Heil, M. Martin, C. McCurdy, R. Rajwar, E. Weglarz, C. Zilles, and M. Lipasti, "Characterizing a Java implementation of TPC-W," in *3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW)*, Toulouse, France, January 2000.

[41] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic Web site benchmarks," in *Proceedings of the 5th Workshop on Workload Characterization*, Austin, Texas, November 2002.

[42] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed, "A publishing system for efficiently creating dynamic Web content," in *IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[43] A. Iyengar and J. Challenger, "Improving Web server performance by caching dynamic data," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[44] Y. Lu, T. F. Abdelzaher, C. Lu, and G. Tao, "An adaptive control framework for QoS guarantees and its application to differentiated caching services," in *International Workshop on Quality of Service (IWQoS)*, 2002.