An Online Optimization-based Technique For Dynamic Resource Allocation in GPS Servers

Abhishek Chandra, **Weibo Gong**†and **Prashant Shenoy**

Department of Computer Science
University of Massachusetts Amherst
{abhishek,shenoy}@cs.umass.edu

†Department of Electrical and Computer Engineering
University of Massachusetts Amherst
gong@ecs.umass.edu

***Abstract*—**

Since web workloads are known to vary dynamically with time, in this paper, we argue that dynamic resource allocation techniques are necessary in the presence of such workloads to provide guarantees to web applications running on shared data centers. To address this issue, we present an analytic model of a server resource that services multiple applications using *generalized processor sharing* (GPS). Using this model, we present online workload prediction and optimization-based techniques to dynamically allocate resources to competing web applications running on shared servers. Our techniques can react to changing workloads by dynamically varying the resource shares of applications, and unlike some prior techniques, can handle the nonlinearity in system behavior. We evaluate our techniques using simulations with synthetic as well as real-world web workloads. Our results show that these techniques can multiplex the resource better than static provisioning, especially under transient overload conditions.

I. Introduction

A. Motivation

The growing popularity of the World Wide Web has led to the advent of Internet data centers that host third-party web applications and services. A typical web application consists of a front-end web server that services HTTP requests, a Java application server that contains the application logic, and a backend database server. In many cases, such applications are housed on managed data centers where the application owner pays for (rents) server resources, and in return, the application is provided guarantees on resource availability and performance. To provide such guarantees, the data center—typically a cluster of servers—must provision sufficient resources to meet application needs. Such provisioning can be based either on a dedicated or a shared model. In the dedicated model, some number of cluster nodes are dedicated to each application and the provisioning technique must determine how many nodes to allocate to the application. In the shared model, which we consider in this paper, an application can share node resources with other applications and the provisioning technique needs to determine how to partition resources on each node among competing applications.

Since node resources are shared, providing guarantees to applications in the shared data center model is more complex. Typically such guarantees are provided by reserving a certain fraction of node resources (CPU, network, disk) for each application. The fraction of the resources allocated to each application depends on the expected workload and the QoS requirements of the application. The workload of web applications is known to vary dynamically over multiple time scales [12] and it is challenging to estimate such workloads a priori (since the workload can be influenced by unanticipated external events—such as a breaking news story—that can cause a surge in the number of requests accessing a web site). Consequently, static allocation of resources to applications is problematic—while over-provisioning resources based on worst case workload estimates can result in potential underutilization of resources, under-provisioning resources can result in violation of guarantees. An alternate approach is to allocate resources to applications dynamically based on the variations in their workloads. In this approach, each application is given a certain minimum share based on coarse-grain estimates of its resource needs; the remaining server capacity is dynamically shared among various applications based on their instantaneous needs. To illustrate, consider two applications that share a server and are allocated 30% of the server resources each; the remaining 40% is then dynamically shared at run-time so as to meet the guarantees provided to each application. Such dynamic resource sharing can yield potential multiplexing gains, while allowing the system to react to unanticipated increases in application load and thereby meet QoS guarantees. Dynamic resource al-

location techniques that can handle changing application workloads in shared data centers is the focus of this paper.

B. Research Contributions

In this paper, we present analytic techniques for dynamic resource allocation in shared servers. We model various server resources using *generalized processor sharing (GPS)* [23] and assume that each application is allocated a certain fraction of a resource. Using a combination of online measurement, prediction and adaptation, our techniques can dynamically determine the resource share of each application based on (i) its QoS (response time) needs and (ii) the observed workload.

We make three specific contributions in this paper. First we present an analytic model of a server resource that supports multiple application-specific queues; each queue represents the workload from an application and is serviced using GPS. Such an abstract GPS-based model is applicable to many server resources—both hardware and software—such as the network interface, the CPU, the disk and socket accept queues. Using this model, we derive an online optimization-based approach to dynamically determine the resource share of each application based on its workload and QoS requirements. An important feature of our optimization-based approach is that it can handle non-linearity in the system behavior.

Determining resource shares of applications using such an online approach is crucially dependent on an accurate estimation of the workload. A second contribution of our work is a prediction algorithm that estimates the workload parameters using online measurements, and uses these parameters to predict the expected load in the near future to enable better resource allocation.

Third, we evaluate the effectiveness of our online prediction and allocation techniques using simulations. We use both synthetic workloads and real-world web traces for our evaluation and show that our techniques adapt to changing workloads fairly efficiently, especially under transient overload conditions.

The rest of the paper is structured as follows. We formulate the problem of dynamic resource allocation in GPS systems in Section II. Section III presents our online prediction and optimization-based techniques for dynamic resource allocation. Results from our experimental evaluation are presented in Section IV. We discuss related work in Section V and present our conclusions in Section VI.

II. Problem Formulation and System Model

In this section, we first present an abstract GPS-based model for a server resource and then formulate the problem of dynamic resource allocation in such a GPS-based system.

A. Resource Model

We model a server resource using a system of n queues, where each queue corresponds to a particular application (or a class of applications) running on the server. Requests within each queue are assumed to be served in FIFO order and the resource capacity C is shared among the queues using GPS. To do so, each queue is assigned a weight and is allocated a resource share in proportion to its weight. Specifically, a queue with a weight w_i is allocated a share $\phi_i = \frac{w_i}{\sum_j w_j}$ (i.e., allocated $(\phi_i \cdot C)$ units of the resource capacity when all queues are backlogged). Several practical instantiations of GPS exist—such as weighted fair queuing (WFQ) [13], self-clocked fair queuing [15], and start-time fair queuing [16]—and any such scheduling algorithm suffices for our purpose. We note that these GPS schedulers are work-conserving—in the event a queue does not utilize its allocated share, the unused capacity is allocated fairly among backlogged queues. Our abstract model is applicable to many hardware and software resources found on a server; hardware resources include the network interface bandwidth, the CPU and in some cases, the disk bandwidth, while software resource include socket accept queues in a web server servicing multiple virtual domains [19], [24].

B. Problem Definition

Consider a shared server that runs multiple third-party applications. Each such application is assumed to specify a desired quality of service; the QoS requirements are specified in terms of a target response time. The goal of the system is to ensure that the mean response time seen by application requests (or some percentile of the response time) is no greater than the desired target response.

In general, each incoming request is serviced by multiple hardware and software resources on the server, such as the CPU, NIC, disk, etc. We assume that the specified target response time is split up into multiple resource-specific response times, one for each such resource. Thus, if each request spends no more than the allocated target on each resource, then the overall target response time for the server will be met.[1]

Since each resource is assumed to be scheduled using GPS, the target response time of each application can be met by allocating a certain share to each application. The resource share of an application will depend not only on the specific response time but also the load in each application. Since the workload seen by an application varies

[1] The problem of how to split the specified server response time into resource-specific response times is beyond the scope of this paper. In this paper, we assume that such resource-specific target response times are given to us.

dynamically, so will its resource share. In particular, we assume that each application is allocated a certain minimum share ϕ_i^{min} of the resource capacity; the remaining capacity $1 - \sum_j \phi_j^{min}$ is dynamically allocated to various applications depending on their current workloads (such that their target response time will be met). Formally, if d_i denotes the target response time of application i and $\bar{T}_i$ is its observed mean response time, then the application should be allocated a share ϕ_i, $\phi_i \geq \phi_i^{min}$, such that $\bar{T}_i \leq d_i$.

Since each resource has a finite capacity and the application workload can exceed capacity during periods of heavy loads, the above goal can not always be met (especially during transient overloads). Consequently, we redefine our system goal to account for overloads as follows. We use the notion of utility to represent the satisfaction of an application based on its current allocation. An application remains satisfied so long as its allocation ϕ_i yields a mean response time $\bar{T}_i$ no greater than the target d_i (i.e., $\bar{T}_i \leq d_i$). The discontent of an application grows as its response time deviates from the target d_i. This discontent can be captured in many different ways. In the simplest case, we can use a piecewise linear function to represent discontent:

$$D_i(\bar{T}_i) = \begin{cases} 0 & \text{if } \bar{T}_i \leq d_i \\ (\bar{T}_i - d_i) & \text{if } \bar{T}_i > d_i \end{cases} \quad (1)$$

In this scenario, the discontent grows linearly when the observed response time deviates from (and exceeds) the specified target d_i. The overall system goal then is to assign a share ϕ_i to each application, $\phi_i \geq \phi_i^{min}$, such that the total system-wide discontent is minimized. That is, the quantity

$$D = \sum_{i=1}^{n} D_i$$

is minimized.

We use this problem definition to derive our dynamic resource allocation mechanism, which is described next.

III. Dynamic Resource Allocation

To perform dynamic resource allocation based on the above formulation, each GPS-based resource on the shared server will need to employ three components: (i) a *monitoring* module that measures the workload seen by each application (e.g., the response time $\bar{T}_i$ needs to be measured to determine the current application discontent), (ii) a *prediction* module that uses the measurements from the monitoring module to estimate the future workload, and (iii) an *allocation* module that uses these workload estimates to determine resource shares such that overall system-wide discontent is minimized. Figure 1 depicts these three components.

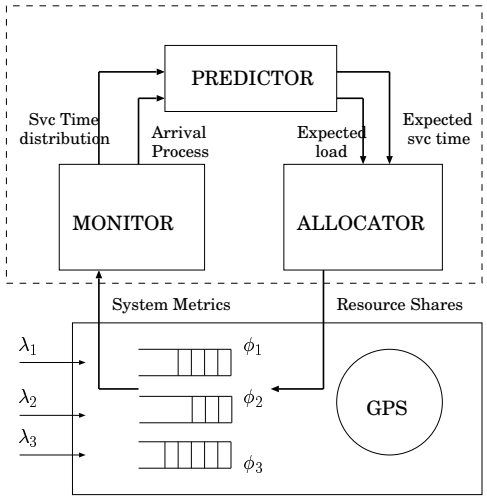


Fig. 1. Dynamic Resource Allocation

In the rest of this section, we first present our dynamic resource allocation techniques and then present mechanisms to predict the future workload based on current observations.

A. Allocating Resource Shares to Applications

The allocation module is invoked periodically to dynamically partition the resource capacity among the various applications running on the shared server. As explained earlier, the share allocated to an application depends on its specified target response time and the estimated workload. We now present an online optimization-based approach to determine resource shares dynamically assuming the workload estimates for each application are known. In the next section, we show how to derive such workload estimates from past observations.

The allocation module needs to determine the resource share ϕ_i for each application such that the total *discontent* $D = \sum_{i=1}^{n} D_i$ is minimized. This problem translates to the following constrained optimization problem:

$$\text{minimize}_{\{\phi_i\}} \sum_{i=1}^{n} D_i(\bar{T}_i)$$

subject to the constraints

$$\sum_{i=1}^{n} \phi_i \leq 1$$

and

$$\phi_i^{min} \leq \phi_i \leq 1.$$

where D_i is a function that represents the discontent of a class based on its current response time $\bar{T}_i$. The two

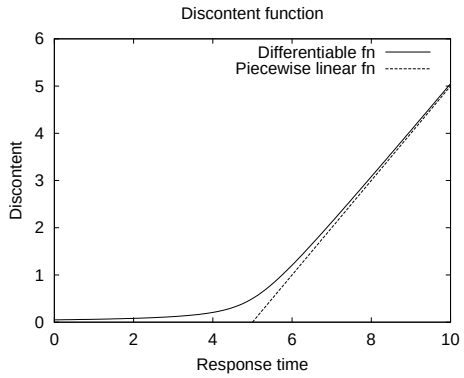


Fig. 2. Two different variants of the discontent function. A piecewise linear function and a continuously differentiable convex functions are shown. The target response time is assumed to be $d_i = 5$.

constraints specify that (i) the total allocation across all applications should not exceed the resource capacity, and (ii) the share of each application can be no smaller than its minimum allocation ϕ_i^{min} and no greater than the resource capacity.

In general, the nature of the discontent function D_i has an impact on the allocations ϕ_i for each application. As shown in Equation 1, a simple discontent function is one where the discontent grows linearly as the response time $\bar{T}_i$ deviates from (and exceeds) the target d_i. Such a D_i, shown in Figure 2, however, is non-differentiable. To make our constrained optimization problem mathematically tractable, we approximate this piece-wise linear D_i by a continuously differentiable function:

$$D_i(\bar{T}_i) = \frac{1}{2}[(\bar{T}_i - d_i) + \sqrt{(\bar{T}_i - d_i)^2 + k}],$$

where $k > 0$ is a constant. Essentially, the above function is a hyperbola with the two piece-wise linear portions as its asymptotes and the constant k governs how closely this hyperbola approximates the piece-wise linear function. Figure 2 depicts the nature of the above function.

The resulting optimization problem can be solved using the Lagrangian multiplier method [8]. In this technique, the constrained optimization problem is transformed into an unconstrained optimization problem where the original discontent function is replaced by the objective function:

$$L(\{\phi_i\}, \beta) = D - \beta \cdot (\sum_{i=1}^{n} \phi_i - 1). \tag{2}$$

The objective function L needs to be minimized subject to the bound constraints on ϕ_i. Here β denotes the shadow price for the resource. Intuitively, each application is charged a price of β per unit resource it uses. Thus, each application attempts to minimize the price it pays for its resource share, while maximizing the utility it derives from that share.

The Lagrangian multiplier method involves solving the following system of partial differential equations

$$\frac{\partial L}{\partial \phi_i} = 0, \quad \forall i = 1, \ldots, n \tag{3}$$

and

$$\frac{\partial L}{\partial \beta} = 0 \tag{4}$$

The solution to this system of equations, derived either using analytical or numerical methods, yields the shares ϕ_i that should be allocated to each application to minimize the system-wide discontent.

We note that the above set of equations are being optimized with respect to ϕ_i, while the discontent function is represented in terms of the response time $\bar{T}_i$. Consequently, we need the relation between response time $\bar{T}_i$ of an application and its resource share ϕ_i, so that D_i can be expressed (rewritten) in terms of ϕ_i in order to solve the above differential equations. We use the queuing dynamics of the system to derive such a relationship next.

A.1 Deriving a relation between $\bar{T}_i$ and ϕ_i

To derive a relation between $\bar{T}_i$ and ϕ_i, let us assume that the adaptation algorithm is invoked every W time units. W is also referred to as the *adaptation window*. Let q_i^0 denote the queue length at the beginning of an adaptation window. Let $\hat{\lambda}_i$ denote the estimated request arrival rate and $\hat{\mu}_i$ denote the estimated service rate in the next adaptation window (i.e., over the next W time units) Then, assuming the values of $\hat{\lambda}_i$ and $\hat{\mu}_i$ are constant, the length of the queue at any instant t within the next adaptation window is given by

$$q_i(t) = \left[q_i^0 + \left(\hat{\lambda}_i - \hat{\mu}_i\right) \cdot t\right]^+, \tag{5}$$

Intuitively, the amount of work queued up at instant t is the sum of the initial queue length and the amount of work arriving in this interval minus the amount of work serviced in this duration. Since the queue length is non-negative, we denote it by x^+, which is an abbreviation for $max(x, 0)$.

Since the resource is modeled as a GPS server, the service rate of an application is effectively $(\phi_i \cdot C)$, where C is the resource capacity, and this rate is continuously available to a backlogged application in any GPS system. Hence, the request service rate is

$$\hat{\mu}_i = \frac{\phi_i \cdot C}{\hat{s}_i}, \tag{6}$$

where $\hat{s}_i$ is the estimated mean service demand per request (such as number of bytes per packet, or CPU cycles per CPU request, etc.).

Note that, due to the work conserving nature of GPS, if some applications do not utilize their allocates shares, then the utilized capacity is fairly redistributed to backlogged applications. Consequently, the queue length computed in Equation 5 assumes a worst-case scenario where all applications are backlogged and each application receives no more than its allocated share (the queue would be smaller if the application received additional unutilized share from other applications).

Given Equation 5, the average queue length over the adaptation window is given by:

$$\bar{q}_i = \frac{1}{W} \int_0^W q_i(t)dt \tag{7}$$

Depending on the particular values of q_i^0, the arrival rate $\hat{\lambda}_i$ and the service rate $\hat{\mu}_i$, the queue may become empty one or more times during an adaptation window. To include only the non-empty periods of the queue when computing $\bar{q}_i$, we consider the following scenarios, based on the assumption of constant $\hat{\mu}_i$ and $\hat{\lambda}_i$:

1) *Queue growth:* If $\hat{\mu}_i < \hat{\lambda}_i$, then the application queue will grow during the adaptation window and the queue will remain non-empty throughout the adaptation window.
2) *Queue depletion:* If $\hat{\mu}_i > \hat{\lambda}_i$, then the queue starts depleting during the adaptation window. The instant t_0 at which the queue becomes empty is given by

$$t_0 = \frac{q_i^0}{\hat{\mu}_i - \hat{\lambda}_i}$$

   If $t_0 < W$, then the queue becomes empty within the adaptation window, otherwise the queue continues to deplete but remains non-empty throughout the window (and is projected to become empty in a subsequent window).
3) *Constant queue length:* If $\hat{\mu}_i = \hat{\lambda}_i$, then the queue length remains fixed ($= q_i^0$) throughout the adaptation window. Hence, the non-empty queue period is either 0 or W depending on the value of q_i^0.

Let us denote the duration within the adaptation window for which the queue is non-empty by W_i (W_i equals either W or t_0 depending on the various scenarios). Then, Equation 7 can be rewritten as

$$\bar{q}_i = \frac{1}{W} \int_0^{W_i} q_i(t)dt \tag{8}$$

$$= \left(\frac{W_i}{W}\right) \left[q_i^0 + \frac{W_i}{2}\left(\hat{\lambda}_i - \hat{\mu}_i\right)\right] \tag{9}$$

Having determined the average queue length over the next adaptation interval, we can derive the average response time $\bar{T}_i$ as the sum of the mean queuing delay and the request service time. We use Little's law to derive the queuing delay from the mean queue length.[2] Thus,

$$\bar{T}_i = \frac{(\bar{q}_i + 1)}{\hat{\mu}_i} \tag{10}$$

Substituting Equation 6 in this expression, we get

$$\bar{T}_i = \left(\frac{\hat{s}_i}{\phi_i \cdot C}\right) \cdot (\bar{q}_i + 1), \tag{11}$$

where $\bar{q}_i$ is as given by equation 9. The values of q_i^0, $\hat{\mu}_i$, $\hat{\lambda}_i$ and $\hat{s}_i$ are obtained using measurement and prediction techniques discussed in the next section.

Using this relation between $\bar{T}_i$ and ϕ_i, we can rewrite the discontent function in terms of the resource share ϕ_i and solve the system of differential equations to obtain values of ϕ_i for the next W time units.

The above optimization-based approach has the following salient features:

- As shown in Equation 11, our techniques assume a non-linear relationship between the response time $\bar{T}_i$ and the resource share ϕ_i, implying a non-linear optimization.
- The share allocated to an application depends on its current workload characteristics ($\hat{\lambda}_i$, $\hat{s}_i$) and the current system state (q_i^0). Consequently, our techniques can operate in an *online* setting and react to sudden (or gradual) changes in the workload on time-scales of tens of seconds or minutes.

B. Workload Prediction Techniques

The online optimization-based allocation technique described in the previous section is crucially dependent on an accurate estimation of the workload likely to appear in each application class. In this section, we present techniques that use past observations to estimate the future workload for an application.

The workload seen by an application can be characterized by two complementary distributions: the *request arrival process* and the *service demand distribution*. Together these distributions enable us to capture the workload intensity and its variability. Our technique measures the various parameters governing these distributions over a certain time period and uses these measurements to predict the workload for the next adaptation window.

[2]Note that the application of Little's Law in this scenario is more accurate when the size of the adaptation window is large compared to the average request service time. Otherwise, this is an approximation.

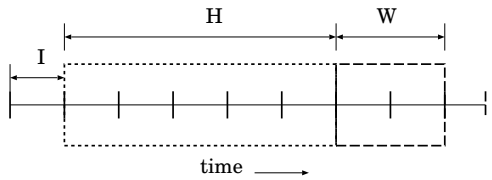


Fig. 3. Time intervals used for monitoring, prediction and allocation

In what follows, we will first present an overview of the monitoring module that is responsible for online measurements and then present techniques for predicting the arrival rate and service demand based on these measurements.

B.1 Online Monitoring and Measurement

The online monitoring module is responsible for measuring various system parameters and workload characteristics. These measurements are based on the following time intervals (see Figure 3):

- *Measurement interval (I):* I is the interval over which various parameters of interest are sampled. For instance, the monitoring module tracks the number of requests arrivals (n_i) in each interval I and records this value.
  The choice of a particular measurement interval depends on the desired responsiveness from the system. If the system needs to react to workload changes on a fine time-scale, then a small value of I (e.g., $I = 1$ second) should be chosen. On the other hand, if the system needs to adapt to long term variations in the workload over time scales of hours or days, then a coarse-grain measurement interval of minutes or tens of minutes may be chosen.
- *History (H):* The history represents a sequence of recorded values for each parameter of interest. Our monitoring module maintains a finite history consisting of the most recent H values for each such parameter; these measurements form the basis for predicting the future values of these parameters.
- *Adaptation Window (W):* As mentioned in the previous section, the adaptation window is the time interval between two successive invocations of the adaptation algorithm. Thus the past measurements are used to predict the workload for the next W time units.

B.2 Estimating the Arrival Rate

The request arrival process corresponds to the workload intensity for an application. The crucial parameter of interest that characterizes the arrival process is the request arrival rate λ_i. An accurate estimate of λ_i allows the allocation module to estimate the average queue length for the next adaptation window.

To do so, the monitoring module measures the number of request arrivals a_i in each measurement interval I. The sequence of these values $\{a_i^m\}$ represents a stochastic process A_i. Since this stochastic process can be non-stationary, instead of trying to compute the rate for the process as a whole, our prediction module attempts to predict the number of arrivals $\hat{n}_i$ for the next adaptation window. The arrival rate for the window, $\hat{\lambda}_i$ is then approximated as $\left(\frac{\hat{n}_i}{W}\right)$ where W is the window length. We represent A_i at any time by the sequence $\{a_i^1, \ldots, a_i^N\}$ of values from the history H.

To predict $\hat{n}_i$, we model the process as an AR(1) process [6] (*autoregressive* of order 1). This is a simple linear regression model in which a sample value is predicted based on the previous sample value [3].

Using the AR(1) model, a sample value of A_i is estimated as

$$\hat{a}_i^{j+1} = \bar{a}_i + R_i(1) \cdot (a_i^j - \bar{a}_i) + e_i^j,$$

where, R_i and $\bar{a}_i$ are the autocorrelation and mean of $\{a_i^m\}$ respectively, and e_i^j is a white noise component. We assume e_i^j to be 0, and a_i^j to be estimated values $\hat{a}_i^j$ for $j \geq N+1$. The autocorrelation R_i is defined as

$$R_i(l) = \frac{E[(a_i^j - \bar{a}_i) \cdot (a_i^{j+l} - \bar{a}_i)]}{\sigma_{a_i}^2}, 0 \leq l \leq N-1, \tag{12}$$

where, σ_{a_i} is the standard deviation of A_i and l is the *lag* between sample values for which the autocorrelation is computed.

Thus, if the adaptation window size is M intervals (i.e., $M = W/I$), then, we first compute $\hat{a}_i^{N+1}, \ldots, \hat{a}_i^{N+M}$ using the AR(1) model, where, $\hat{a}_i^j$ denotes estimated value of a_i for interval j. Then, the estimated number of arrivals in the adaptation window, $\hat{n}_i$, is given by

$$\hat{n}_i = \sum_{j=N+1}^{N+M} \hat{a}_i^j.$$

and

$$\hat{\lambda_i} = \frac{\hat{n}_i}{W}$$

[3] Even though an AR(1) model may not represent A_i accurately, the primary reason for choosing an AR(1) model over more sophisticated models, such as AR(n) models ($n > 1$), ARMA or ARIMA models, is the ease of parameter estimation. Also, we are interested in online estimation, while determination of an appropriate model would require offline post-processing of data or a computationally expensive analysis.

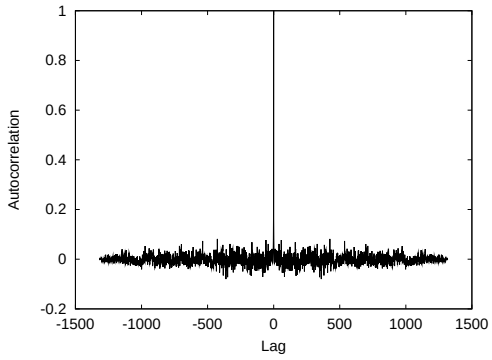


Fig. 4. Autocorrelation of the average service demand time series for a real workload

B.3 Estimating the Service Demand

The service demand of each incoming request represents the load imposed by that request on the resource. Two applications with similar arrival rates but different service demands (e.g., different packet sizes, different per-request CPU demand, etc.) will need to be allocated different resource shares.

To estimate the service demand for an application, the prediction module computes the probability distribution of the per-request service demands. This distribution is represented by a histogram of the per-request service demands over some history. Upon the completion of each request, this histogram is updated with the service demand of that request. The distribution is used to determine the expected request service demand $\hat{s}_i$ for requests in the next adaptation window. $\hat{s}_i$ could be computed as the mean, the median, or a percentile of the distribution obtained from the histogram. For our experiments, we use the mean of the distribution to represent the service demand of application requests.

Note that, unlike the arrival rate, we do not use a regression model for estimating the service demand $\hat{s}_i$. This is because, using service demand values from the recent past does not seem to be indicative of the demands of future requests. To verify this observation, we used the request trace of a real web server (the details of which are given in the next section), and measured the average service demand over fixed-size measurement intervals. We treated the time series of these values as a stochastic process. As shown in figure 4, the autocorrelation values for this process are nearly 0 at all lags, which implies that knowledge of the recent past does not help in estimating future service demands. Hence, it is sufficient to estimate the service demands using a static distribution rather than using an autoregressive stochastic process. We can then use the mean or a high percentile value of the static distribution as our estimate.

B.4 Measuring the Queue Length

A final parameter required by the allocation model is the queue length of each application at the beginning of each adaptation window. Since we are only interested in the instantaneous queue length q_i^0 and not mean values, measuring this parameter is trivial—the monitoring module simply records the number of outstanding requests in each application queue at the beginning of each adaptation period W.

IV. Experimental Evaluation

We demonstrate the efficacy of our prediction and allocation techniques using a simulation study. In what follows, we first present our simulation setup and then our experimental results.

A. Simulation Setup and Workload Characteristics

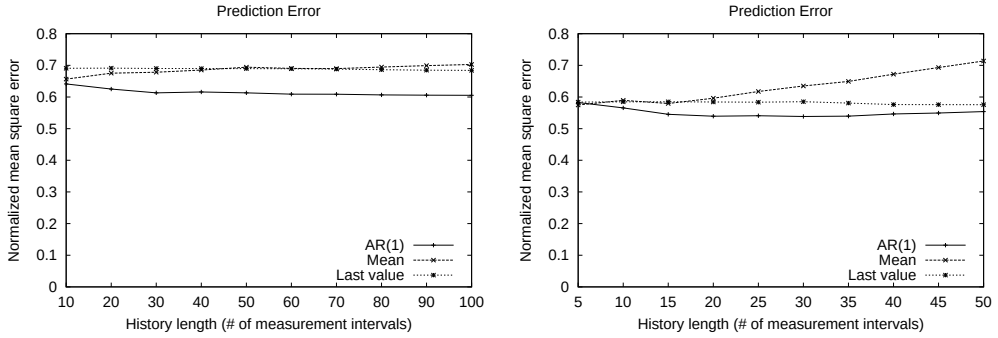
Our simulator models a server resource with multiple application specific queues; the experiments reported in this paper specifically model the network interface on a shared server. We assume that requests in various queues are scheduled using weighted fair queuing—a practical instantiation of GPS. Our simulator is based on the *NetSim* library [17] and *DASSF* simulation package [18]; together these components support network elements such as queues, traffic sources, etc., and provide us the necessary abstractions for implementing our simulator. The adaptation and the prediction algorithms were implemented using *Matlab* [22] (which provides various statistical routines and numerical non-linear optimization algorithms); the Matlab code is invoked directly from the simulator for prediction and adaptation.

We use two types of workload in our study—synthetic and trace-driven. Our synthetic workloads use Poisson request arrivals and assume deterministic request sizes. Our trace workload is based on the World Cup Soccer '98 server logs [4]—a publicly available web server trace. We use a portion of the trace that is 22 hours long and contains a total of 680,645 requests at a mean request arrival rate of 8.6 requests/sec, and a mean request size of 8.83 KB. We use this trace workload to evaluate the efficacy of our prediction and allocation techniques; this workload was also used to determine the correlation between service demands of requests in Section III-B.3 (Figure 4).

Next, we evaluate our prediction techniques and then study our dynamic resource allocation technique.

B. Prediction Accuracy

Our first experiment examines the effectiveness of the prediction algorithm for predicting the arrival rate of requests. As described in section III-B.2, we use an AR(1)



(a) I=1 minute (b) I=5 minutes

Fig. 5. Prediction error comparison

model to predict the number of arrivals for the next adaptation window. We compare this technique to two other prediction mechanisms: (i) prediction using the *mean rate* over the history and (ii) prediction using the *most recent value* of arrival rate. We used the three predictors to estimate the request arrival rate for the World Cup Soccer trace using measurement intervals of $I = 1, 5, 10$ and 20 minutes.

To quantify the prediction accuracy, we use the normalized root mean square error (NRMS) between the predicted and actual trace values as the metric, which is defined as:

$$NRMS = \frac{RMS(\hat{n}_j)}{\sigma_{n_j}},$$

where, RMS is the root mean square error of the predicted values $\{\hat{n}_j\}$ and σ_{n_j} is the standard deviation of the trace values $\{n_j\}$ respectively. This metric indicates how much worse the prediction is compared to the variation in the trace itself. An NRMS value < 1 would indicate that the prediction is better than picking a value randomly from the distribution of trace values.

Figures 5(a) and (b) show these errors for the three predictors for measurement intervals of 1 and 5 minutes respectively, with varying history sizes. As can be seen from the figures, the AR(1) technique has the smallest prediction error. Predicting using the mean arrival rate yields increasing errors with increasing history sizes (since a larger history results in the use of outdated values for prediction). Prediction using the most recent arrival rate yields the same error irrespective of the history size, because it only considers one observation for each estimate. Its error is nevertheless higher than the AR(1) technique (since it does not take into account the long term trend of the arrival process).

Figure 6 depicts the actual arrival rate for the World Cup Soccer workload and the predicted arrival rate using the AR(1) model. As can be seen from the figure, there is a good match between the two values, thereby demonstrating the effectiveness of our prediction technique.

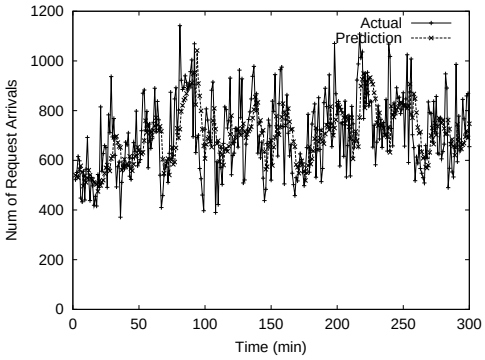


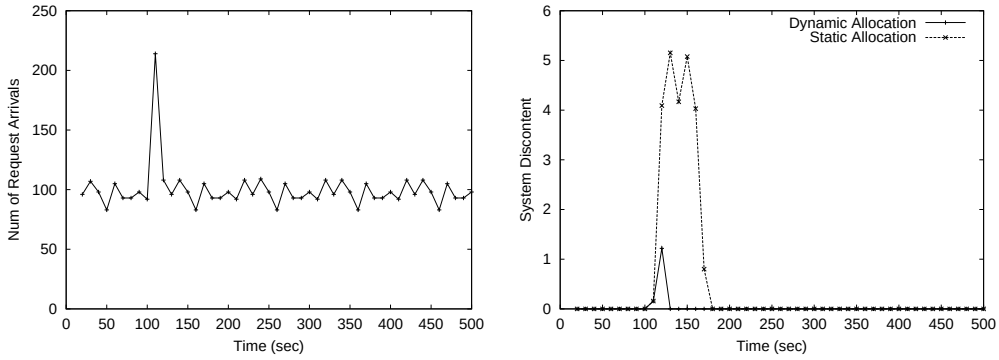
Fig. 6. Actual and Predicted arrival rates

C. Dynamic Resource Allocation

In this section, we evaluate our dynamic resource allocation technique. We conduct two experiments, one with a synthetic web workload and the other with the trace workload and examine the effectiveness of dynamic resource allocation. For purposes of comparison, we repeat each experiment assuming static resource allocation and compare the behavior of the two systems.

C.1 Synthetic Web Workload

To demonstrate the behavior of our system, we considered two web applications that share a server. The benefits of dynamic resource allocation accrue when the workload temporarily exceeds the allocation of an application (resulting in a transient overload). In such a scenario, the dynamic resource allocation technique is able to allocate unused capacity to the overloaded application, and thereby meet its QoS requirements. To demonstrate this property, we conducted a controlled experiment using synthetic web workloads. The workload for each application was generated using Poisson arrivals. The mean



(a) Workload of Application 1 (b) System-wide discontent

Fig. 7. Comparison of static and dynamic resource allocations for a synthetic web workload.

request rate for the two applications were set to 100 requests/s and 200 requests/s. Between time t=100 and 110 sec, we introduced a transient overload for the first application as shown in Figure 7(a). The two applications were initially allocated resources in the proportion 1:2, which corresponds to the average request rates of the two applications. ϕ_{min} was set to 20% of the capacity for both applications and the target delays were set to 2 and 10s, respectively. Figure 7(b) depicts the total discontent of the two applications in the presence of dynamic and static resource allocations. As can be seen from the figure, the dynamic resource allocation technique provides better utility to the two applications when compared to static resource allocation and also recovers faster from the transient overload.

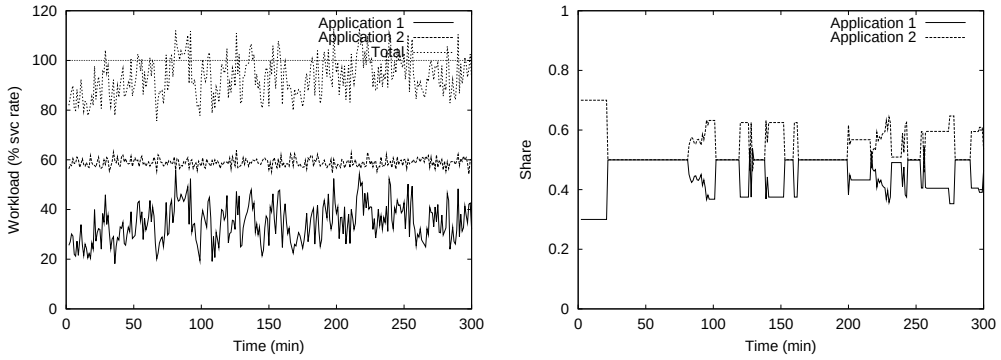
C.2 Trace-driven Web Workloads

Our second experiment considered two web applications. In this case, we use the World Cup trace to generate request arrivals for the first web application; the request size was deterministic (our next experiment examines the impact of heavy-tailed request sizes that are characteristic of this trace). The second application represents a background load for the experiment; its workload was generated using Poisson arrivals and deterministic request sizes. For this experiment, ϕ_{min} was chosen to be 30% for both applications and the initial allocations are set to 30% and 70% for the two applications (the allocations remain fixed for the static case and tend to vary for the dynamic case).

Figure 8(a) shows the workload arrival rate (as a percentage of the resource service rate) for the two applications, and also the total load on the system. As can be seen from the figure, there are brief periods of overload in the system. Figure 8(b) plots the resource share allocated to the two applications by our allocation technique, while Figures 9(a) and (b) show the system discontent values for the dynamic and the static resource allocation scenarios. As can be seen from the figures, transient overloads result in temporary deviations from the desired response times in both cases. However, the dynamic resource allocation technique yields a smaller system-wide discontent, indicating that it is able to use the system capacity more judiciously among the two applications.

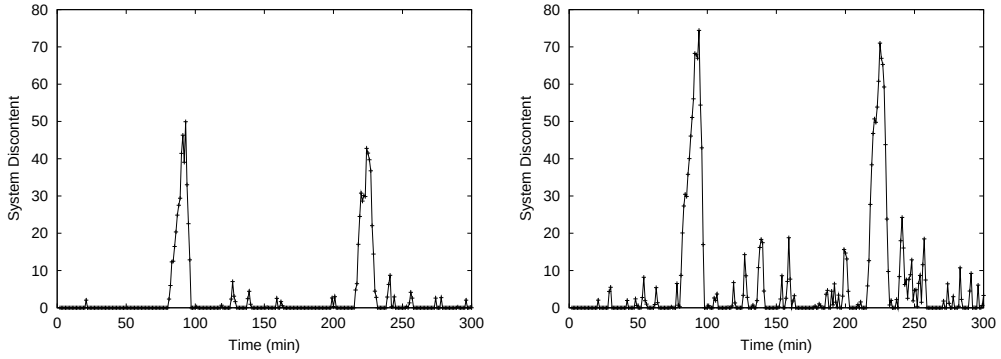
To validate the accuracy of the queuing model used in our allocation algorithm, we compare the system discontent values measured by the simulation to those computed by the model. We conduct the comparison using two different sets of workload parameters as the model inputs: (i) the *actual* parameters as measured by the monitoring component, and (ii) the *predicted* parameters as estimated by the prediction algorithm. The first comparison measures the accuracy of the model with complete knowledge of the actual workload in the system, while the second comparison determines the model performance when coupled with the prediction algorithm. Figures 10 (a) and (b) show the results of these comparisons over a portion of the trace, where the resource was overloaded. As can be seen from the figures, there is a close match between the simulation and the model results in both cases. This demonstrates the fact that the queuing model's estimation is accurate to a large extent using both actual as well as predicted parameters.

Since the above experiment was performed using deterministic request sizes, we repeated the experiment using the actual request sizes from the trace workload (thus, both request arrivals and request service demands were generated using the trace). Note that the request sizes are heavy-tailed, as is common for such workloads. Figure 11 depicts the resulting workload for the two applications and Figure 11(b) plots the resulting resource share allocations for the applications. Figures 12(a) and (b) plot the system-wide discontent for the dynamic and static resource allocation techniques. Similar to the previous experiment, we find that dynamic resource allocation yields lower discon-



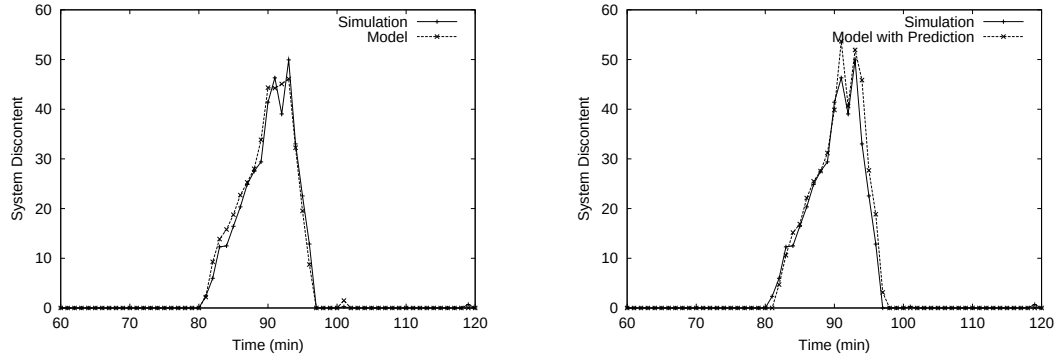
(a) Workload (b) Allocation

Fig. 8. The nature of the workload and the resulting allocations



(a) Dynamic Allocation (b) Static Allocation

Fig. 9. Comparison of static and dynamic resource allocations for a trace web workload.



(a) Model using actual workload parameters (b) Model using predicted workload parameters

Fig. 10. Validation of Queuing Model by comparison to simulation results

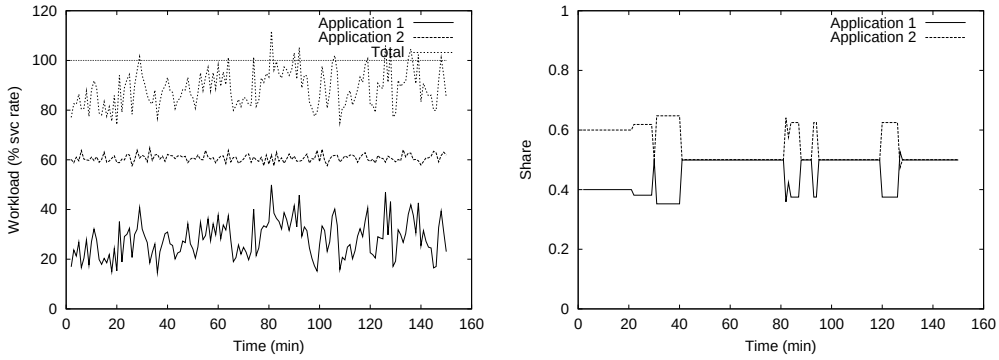
tent values than static allocations.

Together these experiments demonstrate the effectiveness of our prediction and dynamic resource allocation techniques in meeting the QoS requirements of application in the presence of varying workloads.

V. Related Work

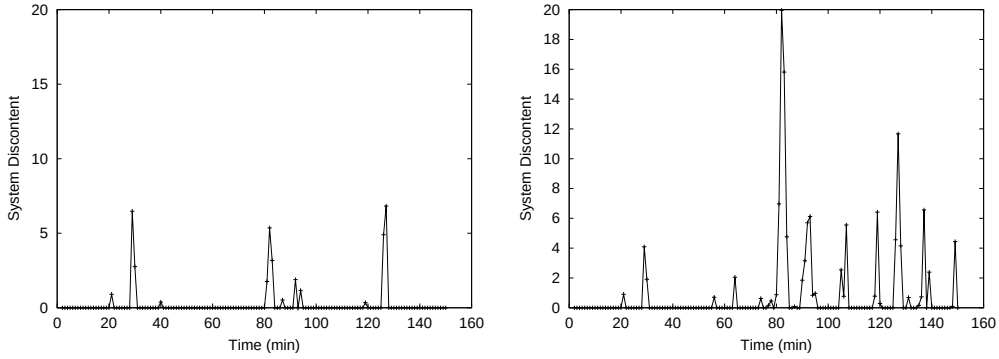
Several research efforts have focused on the design of adaptive systems that can react to workload changes in the context of storage systems [3], [20], general operating systems [26], network services [7], web servers [5], [19], [24], [9], [11] and Internet data centers [2], [25]. In this paper, we focused on an abstract model of a server resource with multiple class-specific queues and presented analytic techniques for dynamic resource allocation; our model and allocation techniques are applicable to many scenarios where the underlying system or resource can be abstracted using GPS.

Recently adaptive techniques for web servers based on a control theoretic approach have been proposed [1], [19], [21], [27]. Most of these techniques (with the exception of [21]) use a pre-determined system model. In contrast,



(a) Workload (b) Allocation

Fig. 11. The workload and the resulting allocations in the presence of varying arrival rates and varying request sizes.



(a) Dynamic Allocation (b) Static Allocation

Fig. 12. Comparison of static and dynamic resource allocations in the presence of heavy-tailed request sizes and varying arrival rates.

our technique focuses on online workload characterization and prediction. Further, these techniques use a linear relationship between the QoS parameter (like target delay) and the control parameter (such as resource share). This is in contrast to our technique that employs a non-linear model derived using the queuing dynamics of the system.

Recently, a queuing model with non-linear optimization has been proposed for scheduling requests in web servers [9]. The primary difference between this paper and the present work is that they use *steady-state queue behavior* to drive the optimization, whereas we use *transient* queue dynamics to control the resource shares of applications. Thus, our goal is to devise a system that can react to transient changes in workload, while they attempt to schedule requests based on the steady-state workload.

Optimization techniques for reducing energy consumption in hosting centers has been studied in [10]. The optimization model attempts to use as few servers as possible to meet the service level agreements for various applications—they maximize the number of unused servers so that they can be powered down to reduce energy consumption.

Two recent efforts have focused on workload-driven allocation in *dedicated* data centers [14], [25]. In these efforts, each application is assumed to run on some number of dedicated servers and the goal is to dynamically allocate and deallocate (entire) servers to applications to handle workload fluctuations. These efforts focus on issues such as how many servers to allocate to an application, how to migrate applications and data, etc., and thus are orthogonal to our present work on *shared* data centers.

VI. Conclusions

In this paper, we argued that dynamic resource allocation techniques are necessary in the presence of dynamically varying workloads to provide guarantees to web applications running on shared data centers. To address this issue, we modeled a shared server resource as a GPS server with multiple queues. Using this model, we proposed an optimization-based allocation technique along with an online workload prediction technique to dynamically allocate resources to competing web applications running on shared servers. Our techniques can react to changing workloads by dynamically varying the resource shares of applications, and unlike some prior techniques, can handle the nonlinearity in system behavior. We evaluated our techniques using simulations with synthetic as well as real-world trace workloads, and showed that these

techniques can multiplex the resource better than static provisioning, especially under transient overload conditions.

As part of future work, we plan to investigate the utility of these techniques for systems employing other types of schedulers (e.g., non-GPS schedulers such as reservation-based). We also intend to implement these techniques in a real system to investigate their performance with real applications. We would also like to explore other optimization techniques using different utility functions and QoS goals.

REFERENCES

[1] 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*, 13(1), January 2002.

[2] J. Aman, C.K. Eilert, D. Emmes, P Yocom, and D. Dillenberger. Adaptive Algorithms for Managing a Distributed Data Processing Workload. *IBM Sytems Journal*, 36(2):242–283, 1997.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles around Storage Administration. In *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[4] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[5] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5), September 1999.

[6] G. Box and G. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.

[7] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, March 1999.

[8] A. Bryson and Y. Ho. *Applied Optimal Control*. Ginn and Company, 1969.

[9] J. Carlström and R. Rom. Application-Aware Admission Control and Scheduling in Web Servers. In *Proceedings of the IEEE Infocom 2002*, June 2002.

[10] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[11] H. Chen and P. Mohapatra. The content and access dynamics of a busy web site: findings and implications. In *Proceedings of the IEEE Infocom 2002*, June 2002.

[12] M R. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12, September 1989.

[14] K Appleby et. al. Oceano - SLA-based Management of a Computing Utility. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, May 2001.

[15] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In *Proceedings of INFOCOM'94*, pages 636–646, April 1994.

[16] P. Goyal, H. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 157–168, August 1996.

[17] B. Liu and D. Figueiredo. Queuing Network Library for SSF Simulator. http://www-net.cs.umass.edu/fluidsim/archive.html, January 2002.

[18] J. Liu and D. M. Nicol. DaSSF 3.0 User's Manual. http://www.cs.dartmouth.edu/∼jasonliu/projects/ssf/docs.html, January 2001.

[19] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. Feedback Control Scheduling in Distributed Systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 2001.

[20] C. Lu, G. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of the Conference on File and Storage Technologies*, January 2002.

[21] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.

[22] Using MATLAB. MathWork, Inc., 1997.

[23] A. Parekh and R. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks – The Single Node Case. In *Proceedings of IEEE INFOCOM '92*, pages 915–924, May 1992.

[24] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An Observation-based Approach Towards Self-Managing Web Servers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.

[25] S. Ranjan, J. Rolia, and E. Knightly H. Fu. QoS-Driven Server Migration for Internet Data Centers. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002)*, May 2002.

[26] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1997.

[27] R. Zhong, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of ICDCS*, July 2002.