Available online at www.sciencedirect.com
SCIENCE DIRECT®
Performance Evaluation 62 (2005) 278–294
PERFORMANCE EVALUATION
An International Journal
www.elsevier.com/locate/peva

Controllable fair queuing for meeting performance goals

Magnus Karlsson[a,*], Christos Karamanolis[a], Jeff Chase[b]

[a] *HP Laboratories, Palo Alto, CA 94304, USA*
[b] *Department of Computer Science, Duke University, Durham, NC 27708, USA*

Available online 10 August 2005

Abstract

Computing and storage utilities must control resource usage to meet contractual performance targets for hosted customers under dynamic conditions, including flash crowds and unexpected resource failures. This paper explores properties of proportional share resource schedulers that are necessary for stability and responsiveness under feedback control. It shows that the fairness properties commonly defined for proportional share schedulers using Weighted Fair Queuing (WFQ) are not preserved across changes to the relative weights of competing request flows. As a result, conventional WFQ schedulers are not *controllable* by a resource controller that adapts by adjusting the weights. The paper defines controllable fairness properties, presents an algorithm to adjust any WFQ scheduler when the weights change, and proves that the algorithm results in controllable-fair schedulers.

The analytic results are confirmed by experimental evaluation using a three-tier Web service and a prototype controllable-fair scheduler called C-SFQ(D). C-SFQ(D) extends concurrency-controlled Start-time Fair Queuing (SFQ(D), which supports proportional sharing in multi-tasking computing resources. The prototype includes an adaptive control system that adjusts the flow weights in C-SFQ(D) to meet latency and throughput targets under a variety of conditions. The experimental results demonstrate the importance of controllable-fair scheduling for feedback control of computing utilities.

Keywords: Weighted fair queueing; QoS; Performance goals; Controllable systems

* Corresponding author.
E-mail address: magnus.karlsson@hp.com (M. Karlsson).
URL: http://www.hpl.hp.com/personal/Magnus_Karlsson (M. Karlsson); http://www.hpl.hp.com/personal/Christos_Karamanolis/ (C. Karamanolis); http://www.cs.duke.edu/%7Echase (J. Chase).

0166-5316/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2005.07.019

1. Introduction

Service providers and enterprises are increasingly hosting services and applications on shared pools of computing and storage resources. For example, in many enterprises, shared network storage servers meet the storage demands of different departments in the organization. Multiplexing workloads onto a shared utility infrastructure allows for on-demand assignment of resources to workloads, which improves resource efficiency while protecting against flash-crowds and outages.

A key problem in such environments is to manage the shared resources in a way that meets the performance requirements of the customers and their workloads, while maximizing the utilization of the resources. The performance goals of customers – response time bound sand minimum throughput requirements – are typically expressed in the form of Service Level Agreements (SLAs). Utility services deploy resource control mechanisms that arbitrate the use of the shared resources to comply with the SLAs of different customer workloads. Depending on the service, workloads may compete for either physical resources (CPU cycles, disk I/O, network bandwidth, etc.) or virtual resources (web server bandwidth, database transactions per second, etc.). Resource control mechanisms include admission control of workloads [1], and throttling or scheduling the demand of individual workloads [2].

This paper focuses on the problem of performance control by varying the shares of resources available to each workload. In particular, we focus on the properties of proportional-share schedulers, which are most commonly implemented using variants of Weighted Fair Queuing (WFQ). The use of WFQ schedulers for meeting SLAs is based on the premise that the performance of a workload varies in a predictable way with the amount of resource available to execute it. A number of publications have reported on the use of proportional sharing for meeting performance goals [3–7].

A key problem for performance control with WFQ schedulers is that, in the general case, certain share assignments do not result in predictable performance, because of the dynamic nature of workloads and systems. To address this problem, resource control mechanisms can use feedback from workload performance to dynamically adjust workload shares in the scheduler. Indeed, a few recent research projects have explored the feasibility of using feedback for resource control [8,9,4,1,2,10]. A challenge, in this case, is to derive the desirable properties for the resulting closed-loop system. Namely, that it is stable (does not oscillate) and that it quickly achieves the desirable performance goals. We thus propose using a rigorous, control-theoretic approach for the design of controllers. We have formalized the problem of meeting performance goals as a quadratic optimization problem which can be solved using off-the-shelf adaptive controllers to tune the scheduler [5].

However, when used in tandem with an adaptive controller, WFQ schedulers could not be controlled to meet performance goals of workloads. The closed-loop system often became unstable and would not converge to the performance goals, even though there were sufficient resources in the system. The main problem is that the obtained performance, given a weight setting, is not predictable. The cause is that the fairness property of the scheduler is not preserved across changes to the weights. This paper proves that WFQ schedulers are unfair in the presence of dynamic control, defines a stronger notion of fairness called controllable fairness that is required to allow stable control, develops a backlog reordering algorithm to ensure that WFQ schedulers are controllable-fair, and proves properties of controllable-fair schedulers.

To validate the notion of controllable-fair scheduling, we developed and implemented a new type of scheduler, CSFQ(D). CSFQ(D) is a controllable-fair variant of concurrency-controlled start-time fair queuing (SFQ(D)), a WFQ algorithm that can deal with concurrency in the scheduled system [4]. We evaluated the scheduler in a three-tier Web application service. The scheduler is placed on the network

path between the service and its clients, where it intercepts http requests sent to the service and re-orders or delays them to enforce proportional sharing of the service's capacity to serve requests. The experimental results show that C-SFQ(D) can be used with an adaptive controller that dynamically sets workload shares in the system to effectively enforce throughput and latency goals in the three-tier service. We also show that when the same controller is used with a conventional WFQ scheduler that is not controllable-fair, it results in an unstable system that oscillates and does not converge to the desired performance goals.

2. Overview

A utility service comprises an ensemble of computing resources (servers, disks, network links, etc.) that are shared by multiple customers with contractual performance assurances (SLAs). SLAs contain statistical performance goals, expressed in terms of averages or percentiles over some time interval. Examples of utility services include shared storage systems [11,4,2,6] or shared clusters hosting a multi-tier Internet application for each customer [12,13]. In all these cases, a number of customer workloads compete for access to the same computational resources. One objective of a utility service is to control the rates by which different workloads use the service, so that the SLAs of the customers are met while maximizing the utilization of the shared resources.

2.1. Resource control

For the discussion in this paper, we generalize the problem of resource sharing, as shown in Fig. 1. The computational unit of resource consumption is called a task. Examples of tasks include I/O operations reaching a disk, threads competing for a CPU, network packets sent over a link, or application-level requests (e.g., http requests) sent to an Internet service. Tasks are grouped into service classes called flows. Examples of flows include all the IO operations accessing a given storage volume, the requests for CPU cycles of a specific virtual machine on a host, or all http requests of the "golden" clients of a Web application.

The objective of a scheduler is to limit the resource consumption of each flow f in proportion to a *weight* ϕ_f assigned to it. If flow weights are normalized to sum to one, then we may interpret each weight

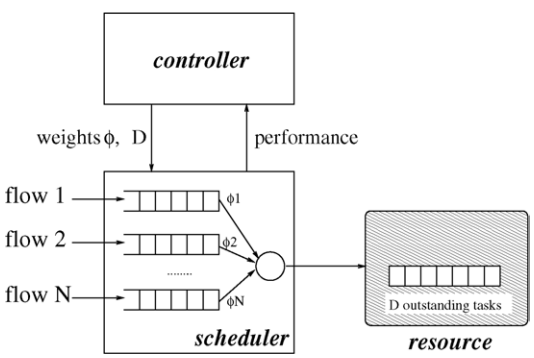

Fig. 1. A task scheduler controls how a resource (physical or virtual) is shared by a number of competing flows. A controller sets the weights of the flows and the degree of concurrency D in the resource, based on feedback about the performance of the flows. The aim is to meet the SLAs of the flows while maximizing the utilization of the resource.

as representing a share of the resources. The weights enforce the essential property of *performance isolation*: they prevent load surges in any flow from unacceptably degrading the performance of another. Thus, weights have to be set to the appropriate values that result in satisfying the SLAs of the different flows. The premise is that the performance of a flow improves when increasing its share of resources. Another factor that affects performance is the degree of concurrency, i.e., the maximum number of concurrent tasks allowed to use the resource at any moment in time. (D in Fig. 1). In general, higher concurrency results in higher aggregate throughput and resource utilization, but also higher response latencies. Thus, the concurrency degree is another scheduler parameter that can be tuned to meet flow SLAs.

We focus specifically on Weighted Fair Queuing (WFQ) schedulers, that have a property known as work conservation. In contrast to guaranteed reservations, which ensure a minimum allotment of resource to each flow even when the flow has low load, a work-conserving proportional-share scheduler shares surplus resources among active flows in proportion to their configured weights. A flow may receive more than its configured share unless the system is fully loaded and all competing flows are *active*, i.e., they have backlogs of queued tasks. The advantage of work-conserving schedulers, that makes them very popular in practice, is that they use resources more efficiently and improve the performance of active flows, when the system is lightly loaded.

2.2. Dynamic control

The flow weights and the degree of concurrency need to be continuously adjusted by a controller in response to observable performance metrics (e.g., response latency, throughput or bandwidth) obtained by each flow. In some cases, the controller may be a human system administrators who adjusts the scheduler parameters manually. In most cases, however, adjustments must happen at a fine time granularity thus requiring automated dynamic control.

The use of dynamic control to meet performance goals has been discussed in the literature [8,9,12,2,1]. A challenge, in this case, is to derive the desirable properties for the resulting closed-loop system. Namely, that it is stable (does not oscillate) and that it achieves the desirable performance goals, preferably fast. Control theory provides an ideal framework for the systematic design of dynamically controlled systems. In particular, existing research has shown that the dynamics of typical computer systems and their workloads require the use of adaptive controllers [2,14]. That is, controllers that automatically adapt their parameters according to the behavior of the target system at any point in time. An example of a computationally efficient adaptively controller, that is widely used in the industry is the *Self-Tuning Regulator* (STR) [15].

The key question now is whether existing computer systems are amenable to dynamic control by such controllers. In our case, in particular, we are concerned with fair-queuing schedulers that arbitrate the sharing of computing services. Control theory has come with a list of necessary and sufficient properties that when met by the controlled system, then a closed loop with an adaptive controller(STR in particular) is stable and converges to the set goal [15,16]:

C.1. The system's behavior must be sufficiently approximated by a linear model. This model must have low variance over time and the relation between actuators and observed behavior must be monotonic in average.

C.2. The system must have a known reaction delay to actuation. That is, there is a known time lag between changing some parameters (e.g., flow weights in our case) and observing the effects of that change.
C.3. Recent actuations must have higher impact than older ones to the behavior of the system. This implies that the effects of an actuation can always be corrected or cancelled by another, later actuation.

Regarding requirement C.1, we have seen in practice, that a linear model provides a good local approximation for the relation between weights and observed performance, given a period for system sampling that is sufficiently long to avoid quantization effects but short enough to trace system dynamics [5]. Moreover, this relation is indeed monotonic in average over long periods of time. However, as we will see in Section 3, existing WFQ schedulers do no meet the low variance requirement as well as properties C.2 and C.3. As a result, native WFQ algorithms are not controllable—they result in unstable systems that do not converge to the desired performance goals as our experimental evaluation in Section 5.3 shows. In Section 4, we propose a variation of WFQ, which we show, both analytically and experimentally, to be controllable.

3. Weighted fair queuing

All existing variants of WFQ scheduling algorithms follow the same principles. Each flow f consists of a sequence of tasks $p_f^0 \ldots p_f^n$ arriving at the server. Each task p_f^i has an associated cost c_f^i bounded by a constant $c_f^{\max}$. Fair queuing allocates the throughput of the resource in proportion to weights assigned to the competing flows. Only the relative values of the weights are significant, but it is convenient to assume that the weight ϕ_f for each flow f represents a percentage share of resource throughput, and that task costs are normalized to a resource throughput of one unit of cost per unit of time.

WFQ schedulers are fair in the sense that active flows share the available resource proportionally to their weights, within some error margin that is bounded by a constant over any time interval. Formally, if $W_f(i)$ is the aggregate cost of the tasks from flow f served during any time interval i, then a fair scheduler guarantees:

$$\left| \frac{W_f(i)}{\phi_f} - \frac{W_g(i)}{\phi_g} \right| \leq U_{f,g}, \tag{1}$$

where f and g are any two flows continuously backlogged with tasks during i. Interval $i = [t_i, t_{i+1})$ is the time period between the ith and $i+1$th sampling/actuation in the system. $U_{f,g}$ is a constant that depends on the flow weights and the maximum cost of flow tasks. All algorithms try to ensure low values of $U_{f,g}$, which indicates better fairness.[1] Poor fairness implies large variability in the relation between performance and weights, violating property C.1 in Section 2.2.

WFQ schedulers dispatch tasks in order of *tags* assigned at task arrival time. When the jth task p_f^j of flow f arrives, it is assigned a start tag $S(p_f^j)$ and a finish tag $F(p_f^j)$ as follows:

$$F(p_f^0) = 0, \tag{2}$$

$$S(p_f^j) = \max\left(v(A(p_f^j)), F(p_f^{j-1})\right), \quad j \geq 1, \tag{3}$$

[1] Fairness is always defined on all pairs of flows; of course, there may be more than two flows using a resource.

$$F(p_f^j) = S(p_f^j) + \frac{c_f^j}{\phi_f}, \quad j \geq 1, \tag{4}$$

where $A(p_f^j)$ is the arrival time of task p_f^j. Table 1 summarizes the symbols used in this paper.

These tags represent the times at which each task should start and finish according to a scheduler notion of virtual time $v(t)$. Virtual time advances monotonically and is identical to real time under ideal conditions. Calculating $v(t)$ exactly is computationally expensive, but there are two WFQ algorithms that approximate $v(t)$ efficiently by clocking the rate at which the resource actually completes work. Self Clocked Fair Queuing (SCFQ) [17] and Start-time Fair Queuing [18] (SFQ) approximate $v(t)$ with (respectively) the finish tag or start tag of the task in service at time t.

All WFQ algorithms have well-defined fairness bounds. SCFQ [17] and SFQ [18] are among the algorithms with the best known bound:

$$U_{f,g} = \left(\frac{c_f^{\max}}{\phi_f} + \frac{c_g^{\max}}{\phi_g} \right). \tag{5}$$

Most existing WFQ schedulers are designed for resources that handle one task at a time, such as a router's outgoing link or a CPU, and thus are not suitable for all computing resources (e.g., disk, multi-processors, file servers). A *concurrency-controlled* variant of SFQ (SFQ(D)) has been proposed recently to deal with task scheduling in multi-tasking resources [4]. In this case, $U_{f,g}$ also depends on the maximum concurrency D allowed in the resources. D reflects a trade-off between resource utilization and the worst-case fairness bound of the scheduler.

3.1. WFQ is not controllable

In this section, we show that, in the general case, WFQ algorithms cannot ensure any fairness bound under dynamic control of the weights. For the proofs, we refer to WFQ algorithms that emulate $v(t)$ by the start tag of the last submitted task (e.g., SFQ [18]). The proofs are similar for algorithms that emulate

Table 1
Frequently used symbols in this paper

Symbol	Meaning
$\phi_f(i)$	Weight of flow f during time interval i
p_f^j	The j-th task of flow f
c_f^j	Cost of task p_f^j
$c_f^{\max}(i)$	Maximum cost for a task from flow f during time interval i
$v(t)$	Virtual time at time t
$D(i)$	The maximum number of outstanding tasks during time interval i
$D'(i)$	The actual number of outstanding tasks during time interval i
$A(p_f^j)$	Arrival time of task p_f^j
$S(p_f^j)$	Start tag of task p_f^j
$F(p_f^j)$	Finish tag of task p_f^j
$W_f(i)$	Total amount of work/cost served from flow f during time interval i
$U_{f,g}(i)$	The fairness bound during time interval i
$U_{f,g}^*$	The controllable fairness bound over a sequence of time intervals

$v(t)$ by the finish tag of the last submitted task (e.g., SCFQ [17]), but are omitted due to lack of space. We assume without loss of generality that all tasks have unit cost.

The proven fairness bounds for WFQ schedulers assume that flow weights are fixed over time. By the definition of interval, the values of flow weights do not change for the duration of an interval. However, when weights do change dynamically between intervals, i.e., $\phi_f(i) \neq \phi_f(i-1)$ for some flow f, then there may exist intervals in which a flow receives no service irrespectively of its weight setting. Consider the example of Fig. 2 with two continuously backlogged flows f and g. Suppose that during time interval 1 $\phi_f = 0.01$ and $\phi_g = 1$. f has one task served, and the start tag of the next task is set to $v(t) = 100$, as $S(p_f^2) = \max(0, 0 + 1/0.01) = 100$ according to (3). Flow g has the higher weight, so it has two tasks served. Thus, by the end of interval 1, $v(t) = 2$. At the beginning of interval 2, the weights are changed to $\phi_f = 0.5$ and $\phi_g = 0.5$. Yet not a single task from f is processed, as $S(p_f^2) = 100$, well ahead of $v(t)$. In fact, even a $\phi_f = \infty$ would produce exactly the same result, as the start tag of the second task of f was computed using the weight during interval 1.

This counter example shows that, when flow weights change, there exist intervals during which the bound of (5) does not hold. In other words, fairness may be arbitrarily bad in any single interval. That is, flow performance may vary arbitrarily depending on past weight settings and flow activity. Thus, property C.1 is violated.

The root cause of this problem is that the tags of the backlogged tasks are not adjusted in any way when the weights change. In order to improve the fairness of WFQ schedulers, we need to recalculate the tags of backlogged tasks when weights change. One naive way of doing this would be to use Eqs. (2)–(4) to recompute the tags of all backlogged tasks, every time the weights change. All flows would start with a clean slate ($F(p_f^0) = 0$) for this interval. This re-computation would indeed result in a good fairness bound for every single interval i, as given by Eq. (5). Thus, property C.1 would be satisfied. However, as we prove in the following theorem, using this approach for tag re-computation does not provide a fairness bound when looking over a sequence of time intervals. As a consequence, properties C.2 and C.3 in Section 2.2 are still not satisfied. But, first, we introduce the notion of controllable fairness to capture the fairness of a scheduler when weights vary dynamically.

Definition 1. For any sequence of consecutive intervals $T = \langle i, \cdots, i + N - 1\rangle$ during which flows f and g are constantly backlogged and weights $\phi_f(i)$ and $\phi_g(i)$ are constant within each interval i, $i \in T$, controllable fairness is defined as:

$$\sum_{i \in T} \left| \frac{W_f(i)}{\phi_f(i)} - \frac{W_g(i)}{\phi_g(i)} \right| \leq U^*_{f,g}. \tag{6}$$

Here, $U^*_{f,g}$ is the controllable fairness bound for the entire sequence of intervals.

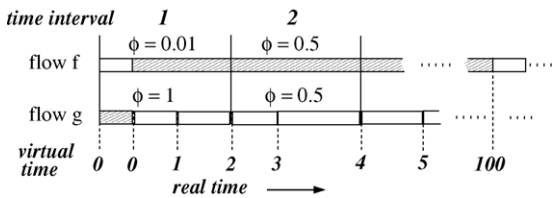

Fig. 2. When weights change in SFQ, there exist intervals in which a flow receives no service independent of its weight setting. The white blocks depict task execution; the gray blocks depict backlog but no execution.

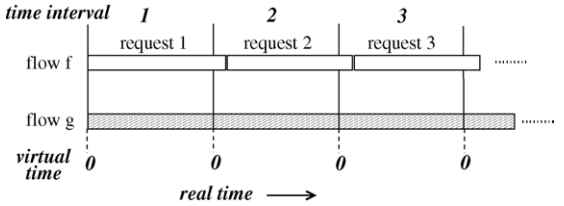

Fig. 3. Example showing that it is possible to construct an unbounded number of consecutive intervals during which there is a flow that receives no service, even though it has non-zero weight.

Theorem 1. *When the flow weights can vary and the tags of backlogged tasks are recomputed using Eqs. (2)–(4) every time some weights change, then the controllable fairness is unbounded, i.e.,* $U^*_{f,g} = \infty$.

Proof. We use a counter-example to show that $U^*_{f,g}$ is unbounded as $N \to \infty$. Consider the scenario of Fig. 3. There are two flows f and g that are continuously backlogged during an infinite sequence of intervals. The start tags of the first tasks of both flows are set to 0 at the beginning of interval 1. WFQ arbitrarily picks to submit the task of f. At the beginning of interval 2 the weights are changed to some new value (the actual value does not matter in this example). At this point, the task of flow f is still not completed, thus, virtual time is still 0. After recomputing the tags of the backlogged tasks, the start tags of the next tasks of both flows are again 0. The outstanding task from f completes during interval 2 and WFQ arbitrarily picks to submit the next task from flow f, as both have the same start tag. This pattern of execution and tag re-computation may continue for an infinite sequence of intervals. Flow f receives all the resource, while g gets nothing. Theorem 1 follows directly from the above counter-example. □

We thus need a tag re-computation phase that results in bounded controllable fairness as $N \to \infty$. A scheduler with this property is presented in the following section. The discussion and counter examples used in this section refer to resources that process one task at a time. The results are trivially applicable to concurrency-controlled WFQ variants (e.g., SFQ(D) [4]).

4. Controllable WFQ

In this section we propose an extension to WFQ algorithms that provably provides good controllable fairness and thus good predictability and responsiveness when flow weights change. In particular, we present and analyze an algorithm called Controllable SFQ, CSFQ for short, which is an extension of SFQ. The extension is also applicable to finish-tag emulated algorithms.

With C-SFQ, the following recursive computation is performed whenever any weights change. The computation updates the tags of the backlogged requests of the flows for which the weight have changed. Assume, without loss of generality, that there are Q_f backlogged requests for flow f and that they are numbered from j to $j + Q_f - 1$. In the following equations, i is the new interval. $v(t)$ refers to the value of virtual time as it evolved in previous intervals, according to WFQ.

$$F(p_f^{j-1}) = S(p_f^{j-1}) + \frac{c_f^{j-1}}{\phi_f(i)}, \tag{7}$$

$$S(p_f^k) = \max\left(v(t), F(p_f^{k-1})\right), \quad j \le k < j + Q_f, \tag{8}$$

$$F(p_f^k) = S(p_f^k) + \frac{c_f^k}{\phi_f(i)}, \quad j \le k < j + Q_f. \tag{9}$$

Eq. (7) recomputes the finish tag of the last submitted request of flow f (in some interval before i), as if it had the new weight setting. The tags of the backlogged requests are adjusted accordingly in Eqs. (8) and (9) which are equivalent to (3) and (4) of WFQ. Re-computation (7) moves the start tag of the next request of f further down in time if the weight has decreased, and closer in time if it has increased. When the weights have not changed, this algorithm reduces to the original WFQ algorithm.

The intuition behind the following theorem is that C-SFQ behaves exactly like SFQ within each interval. Thus, the fairness bound within every single interval is the same as that of SFQ.

Theorem 2. *For any sequence T of consecutive intervals during which flows f and g are constantly backlogged, the controllable fairness of C-SFQ is bounded by*:

$$U_{f,g}^* = \max_{i \in T} \left(\frac{c_f^{\max}(i)}{\phi_f(i)} + \frac{c_g^{\max}(i)}{\phi_g(i)} \right). \tag{10}$$

Proof. Assume that for each interval i there is a hypothetical SFQ execution, such that all the following apply:

(1) For every flow f, the weight of this execution is constant throughout the execution and equal to the C-SFQ weight during i, i.e., $\phi_f^s = \phi_f^c(i)$ for all f. (2) At some point in time, the virtual time of the SFQ execution is equal to that of C-SFQ at the beginning of interval i, i.e., $v^s(t') = v^c(t)$. (3) At that same point in time, the finish tag of the last submitted request in the SFQ execution is equal to the re-calculated finish tag by C-SFQ at the beginning of interval i, $F^s(p_f^k) = F^c(p_f^{j-1})$ for some k and j. (4) At that same point in time, the set of backlogged requests for all flows in the SFQ execution is the same as that in the CSFQ case. (5) From that point in time and at least for a period of time equal to that of interval i, the SFQ scheduler receives the same sequence of requests as C-SFQ.

If C-SFQ executes M steps in interval i, all those steps would be identical to the M following steps in the SFQ execution. Thus, the fairness bound of C-SFQ during interval i would be the same as that of SFQ for the same M steps.

We now need to show that it is always possible to construct a sequence of requests for a hypothetical SFQ so that all the above hold. It is trivial to construct such an execution using SFQ, by submitting a request with cost $c_f^k = F^c(p_f^{j-1})\phi_f^s$, where $\phi_f^s = \phi_f^c(i)$. This ensures that $F^s(p_f^k) = c_f^k/\phi^s = F^c(p_f^{j-1})\phi^c(i)/\phi^c(i) = F^c(p_f^{j-1})$. If at that point $v^c(t) > F^c(p_f^{j-1})$, then $v^s(t')$ can be advanced to $v^c(t)$ by sending one request from flow g where the ratio $c_g/\phi_g = v^c(t) - v^s(t')$. We do not need to consider the case where $v^c(t) \le F^c(p_f^{j-1})$, as the max expression in (8) favors the $F^c(p_f^{j-1})$ term. If at this point, SFQ instantaneously receives the same set of requests as those backlogged in the C-SFQ case at the beginning of i, their backlogged requests will have the exact same start and finish tags.

We know that for any period of time $[t_1, t_2)$, SFQ ensures fairness bounded by $U_{f,g} = \left(\frac{c_f^{\max}}{\phi_f} + \frac{c_g^{\max}}{\phi_g}\right)$ [18]. Thus, this bound holds for every single interval of an execution with C-SFQ. In fact, the fairness bound in every single interval is a function of the maximum cost of the requests actually executed during that interval (not of the maximum cost of any request of a flow). This results in a tighter fairness bound

for each interval i, defined as:

$$U^*_{f,g}(i) = \left(\frac{c^{\max}_f(i)}{\phi_f(i)} + \frac{c^{\max}_g(i)}{\phi_g(i)} \right). \tag{11}$$

Thus, the fairness bound across a sequence of intervals is the worst bound among all individual intervals in the sequence, given by Eq. (10). □

As shown in Section 5.2, the maximum concurrency D also needs to be adjusted according to system and workload dynamics. A controllable scheduler must thus be fair even when D changes. We present and analyze an algorithm called Controllable SFQ(D), C-SFQ(D) for short, which is an extension of SFQ(D) [4]. The original fairness bound for SFQ(D) for when weights and D do not change is [4]:

$$U_{f,g} = (D+1)\left(\frac{c^{\max}_f}{\phi_f} + \frac{c^{\max}_g}{\phi_g} \right). \tag{12}$$

Theorem 3 provides the controllable fairness bound for C-SFQ(D) when D as well as flow weights change between intervals. To provide that bound we first prove the following lemma.

Lemma 1. *The number of outstanding requests during interval i, denoted $D'(i)$ is bounded by*:

$$D'_{\max}(i) = \max(D(i), D(j)), \tag{13}$$

where $D(0) = 0$ and j, $j < i$ is the latest interval before i during which a request was dispatched to the service.

Proof. Consider a sequence of intervals during which all flows are constantly backlogged. Interval $j < i$ is the last interval before i during which at least one requests is dispatched. That means that the number of outstanding requests during j is $D'(j) = D(j)$. On the other hand, no requests are dispatched during any interval between j and i. That is, the number of outstanding requests in all these intervals is $D'(k) = D(j)$, for all $j \le k < i$. There are two cases to consider for interval i:

(1) If $D'(k) \le D(i)$, there are $D(i) - D'(k)$ new requests that the scheduler can dispatch to the service in i. Thus, the maximum possible number of outstanding requests during i is $D'_{\max}(i) = D(i)$ as the flows are continuously backlogged.
(2) If $D'(k) > D(i)$, a new request can be submitted only after $D'(k) - D(i) + 1$ requests have completed. Thus, the largest possible $D'(i)$ occurs when no request is completed in interval i. That is, the maximum possible number of outstanding requests during i is $D'_{\max}(i) = D'(k) = D(j)$.

In either case, $D'_{\max}(i)$ is independent of any $D(m)$, $m < j$. □

Theorem 3. *For any sequence T of consecutive intervals during which flows f and g are constantly backlogged and both D and flow weights vary between intervals, the controllable fairness of C-SFQ(D) is bounded by*:

$$U^*_{f,g} = \max_{i \in T} \left((D'_{\max}(i) + 1) \left(\frac{c^{\max}_f(i)}{\phi_f(i)} + \frac{c^{\max}_g(i)}{\phi_g(i)} \right) \right), \tag{14}$$

where $D'_{\max}(i)$ is defined as in Lemma 1.

Proof. When the maximum concurrency D is changed between intervals, the maximum possible number of pending requests during some interval i is given by $D'_{\max}(i)$ in Eq. (13). According to (12), the bound for a specific interval i is then $(D'_{max}(i)+1)\left(\frac{c_f^{\max}(i)}{\phi_f(i)}+\frac{c_g^{\max}(i)}{\phi_g(i)}\right)$. Thus, the worst-case bound in sequence T is the highest bound of any single interval $i \in T$, as given by Eq. (14). □

In C-SFQ(D), we now have a scheduler that is controllable, i.e., it provably satisfies all the requirements stipulated in Section 2.2.

5. Experimental evaluation

In this section, we present experimental results from a real system, that reconfirm the analytical results of earlier sections. In particular, we make the following points.

- Demonstrate that the values of flow weights and D have to vary dynamically in order to enforce performance goals, given the dynamics of a realistic system and its workloads.
- Show that a typical WFQ scheduler, SFQ(D), is not controllable in practice, when flow weights vary dynamically.
- Confirm that the proposed WFQ extension results in fair, controllable schedulers for varying weights and D.
- Perform a sensitivity analysis of the controllable fairness of C-SFQ(D), with respect to the values as well as the deltas of weights and D.

5.1. Experimental platform

We use a three-tier system as our platform for all the experiments in this section. According to the terminology of the previous sections, the entire three-tier service is the shared resource we are concerned with. The system consists of three components: a web server, an application server and a database server. A controlled scheduler is placed on the network path between the clients and the service front end (web server). Client requests are intercepted by the scheduler which aims at enforcing proportional sharing. The scheduler forwards the requests to the web server and, unless they are for static content, they are forwarded to the application server. The application tier generates a dynamic page from information it obtains from the database server. The application server then forwards the generated page to the web server, which, in turn, responds to the client that requested it. Responses are also intercepted by the scheduler for keeping performance statistics.

The web, application and database servers are hosted on separate server blades, each with two 1 GHz Pentium III processors, 2 GB of RAM, one 46 GB 15 krpm SCSI Ultra160 disk, and two 100 Mbps Ethernet cards. The web server is Apache version 2.0.48 with a BEA WebLogic plug-in. The application server is BEA WebLogic 7.0 SP4 over Java SDK version 1.3.1 from Sun. The database client and server are Oracle 9iR2. All three tiers run on Windows 2000 Server SP4. The site hosted on the three-tier system is a version of the Java PetStore (`java.sun.com`) that has been tuned in order to support a large number of concurrent users.

The workload applied to this system mimics real-world user behavior [19] on the PetStore shopping site. These users log in, browse and search for products, put products in their carts, and sometimes checkout

the cart which gives rise to credit card verifications, adjusting inventory, etc. The end-to-end latencies vary between 10 and 700 ms for the various operations. The workload also captures the corresponding time scales and probabilities these occur with. This workload is generated using `httperf` on a separate machine that is identical to the ones above but runs Linux. The scheduler and controller also run on this machine. For the experiments in the rest of this section, we generate 80 concurrent client sessions and we usually consider two flows each consisting of 40 clients. The sample interval for gathering statistics and for changing the weights and D is 1 s.

5.2. Weights and concurrency degree need to vary continuously

This section demonstrates that the weights and the concurrency degree (D) in a scheduler have to be continuously adjusted to meet performance goals. There are a number of reasons for this: variation to the service capacity, changes to the number of clients accessing the service, and modifications to the service's hardware or software.

For example, consider the scenario of the left graph in Fig. 4. There are a number of flows accessing the system, but we show the performance of only one of these. This flow has a latency goal of 60 ms. At different instances during the depicted run, different weights are needed to meet that latency goal. At some points a weight of 0.3 is sufficient, while at different points even a weight of 0.7 is not enough. If a weight of 0.7 would have been chosen constantly, the latency goal would have been met most of the time by a wide margin, thus wasting valuable resources that other flows could have used. It is thus desirable to adjust the weights dynamically in reaction to the obtained performance. Note, that this experiment was performed on a dedicated system and network, with a constant number of clients accessing a single application that does not change. Even under these ideal condition, we see a lot of variation in the latency provided by the system due to its unpredictability.

The degree of concurrency D needs also to be dynamically adjusted, for the same reasons. We demonstrate this need with the example of the right graph in Fig. 4 where we have a flow with an end-to-end latency goal of 50 ms. Before 60 s, either value of D (2 or 16) can achieve the goal most of the time. In this case, a $D = 16$ should be preferred, as it provides higher system throughput. But after 60 s, $D = 16$ cannot meet the latency goal any longer, thus it needs to get adjusted down to 2 to meet the goal again.

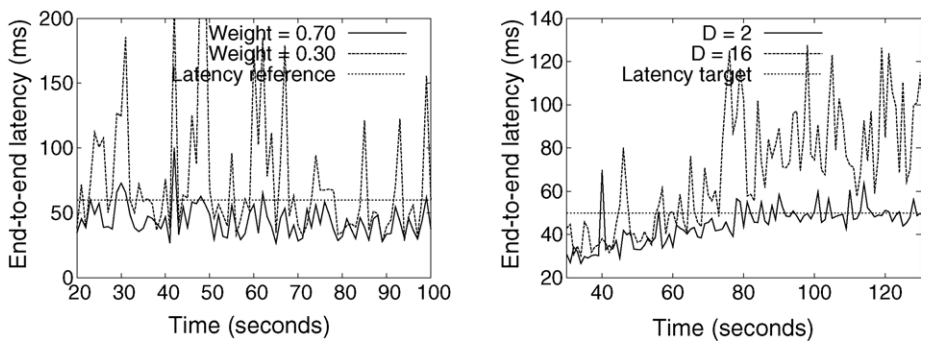

Fig. 4. Demonstrates the need to rapidly adjust flow weights and D. The graph on the left shows how two different weights values produce very different end-to-end latencies for a flow. The graph on the right shows how two different settings of D result in different latencies.

5.3. Fairness of SFQ(D) and C-SFQ(D)

In this section, we evaluate the effects on controllability due to the proposed tag recomputation algorithm. In particular, we focus on the effective differentiation achieved by SFQ(D) and C-SFQ(D) and how this varies with changes in the values of flow weights and D. We use the following metric, called unfairness (Υ), to quantify effective differentiation.

Definition 2. For a set of flows F, the unfairness of the resource allotment to flow $f \in F$ compared to all other flows during interval i is captured by:

$$\Upsilon(f) = 100 \left| \frac{\phi_f(i)}{\sum_{g \in F} \phi_g(i)} - \frac{W_f(i)}{\sum_{g \in F} W_g(i)} \right|. \tag{15}$$

An unfairness of 0 means that the scheduler is perfectly fair and provides perfect differentiation, while an unfairness of 100 signifies no differentiation at all. The higher the unfairness exhibited by a scheduler, the harder it is to control that scheduler for enforcing performance goals.

First, we analyze the effects of the weight values on unfairness. Our goal is to capture the effects of the entire spectrum of possible weight values (a controller could set a weight of any value) without skewing the results due to a specific algorithm of changing the weights between intervals (e.g., a smooth gradual change could have a lesser impact on unfairness). Thus, we set the flow weights randomly using a white noise generator. Unfairness results with an actual controller are reported in Section 5.4. For the experiments reported in Fig. 5, we consider runs with two flows, f and g. At every sample interval, ϕ_f is set to a random number uniformly drawn from the interval $(0, 100)$; the weight of flow g is set to $\phi_g = 100 - \phi_f$. The maximum concurrency in the service is set to 4. The results are only shown for flow f, as the same conclusions can be drawn from the results of flow g. The left graph in Fig. 5 shows that the unfairness of C-SFQ(D) is approximately two orders of magnitude lower than that for SFQ(D). About 99% of the sampled intervals have an unfairness of less than 1 for C-SFQ(D), while for SFQ(D) 90% of the intervals have an unfairness higher than 1. Indeed, 40% of those intervals have an unfairness higher than 25, which we will see later in Section 5.4 results in an uncontrollable system.

But it is not just the absolute weight values that affect unfairness. The example of Fig. 2 showed that the larger the relative change, the longer it takes for weight settings to have any effect on the flows resulting in higher unfairness. To examine how a *weight delta* (ϕ_Δ) affects the unfairness of a scheduler, we have

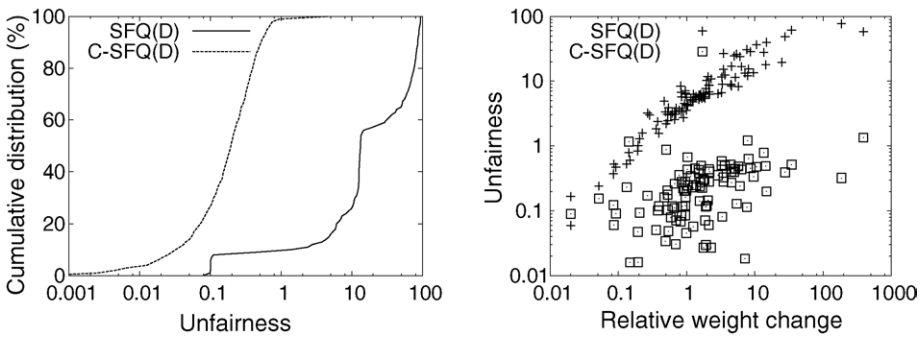

Fig. 5. Sensitivity analysis of the unfairness $\Upsilon(f)$ of SFQ(D) and C-SFQ(D) against weight settings and changes during a run of 1000 intervals. The graph on the left shows the CDF of unfairness for all the intervals of a run (the x-axis is in logarithmic scale). The graph on the right shows unfairness against the relative weight changes (both axes are in logarithmic scale).

plotted the unfairness $\Upsilon(f)$ of SFQ(D) and C-SFQ(D) as a function of relative weight change, in the right graph of Fig. 5. The results come from the same experiment as in the left graph. Relative weight change in the case of two workloads f and g is defined as:

$$\phi_\Delta(i) = \frac{1}{2}\left(\frac{|\phi_f(i) - \phi_f(i-1)|}{\phi_f(i-1)} + \frac{|\phi_g(i) - \phi_g(i-1)|}{\phi_g(i-1)}\right). \tag{16}$$

We see that the unfairness of C-SFQ(D) is approximately two orders of magnitude below that of SFQ(D), for an average change of 10% or more, i.e., $\phi_\Delta(i) \geq 0.1$. As we have seen in Section 5.2, weight changes of that magnitude are not uncommon in real systems. The unfairness of C-SFQ(D) suffers no substantial degradation as the relative weight change increases. Thus, C-SFQ(D) can be safely used even with aggressive controllers.

Let us now examine how D, the degree of concurrency allowed in the service, affects the unfairness of a scheduler. The focus is on C-SFQ(D), as we have just shown that SFQ(D) is unfair irrespective of D. Fig. 6 shows the unfairness as a function of D. The weights during each interval are still picked randomly using the process described previously. From the graph on the left, we can see that unfairness increases with the value of D, as expected from Theorem 3. Up until $D = 16$, unfairness increases by less than 100% at each data point, but at $D = 32$ it jumps up by one order of a magnitude. This is due to the effects of work-conservation kicking in somewhere between $D = 16$ and $D = 32$. That is, D is high enough that not all flows remain backlogged constantly. When this occurs, the scheduler purposefully violates the fairness condition in order to use the system efficiently.

On one hand, work-conservation is a desired property as it increases the total throughput of the system and the utilization of service resources. On the other hand, it has a negative effect on the ability to control the system. Thus, the value of D must be chosen so that the system is operating at nearly full capacity while unfairness is low at the same time. The right graph of Fig. 6 plots the total throughput of the system as a function of D. We can see that $D = 8$ or $D = 16$ provide a good trade off between throughput and unfairness. Another issue to be considered is the effects of D on end-to-end latency, i.e., the response delay perceived by the clients of the service. As the right graph in Fig. 6 shows, the median end-to-end latency goes down as D increases, due to the parallelism inside the three-tier system. It reaches its minimum value at $D \geq 8$. This is compatible with the values for D derived from the trade-off between unfairness and throughput.

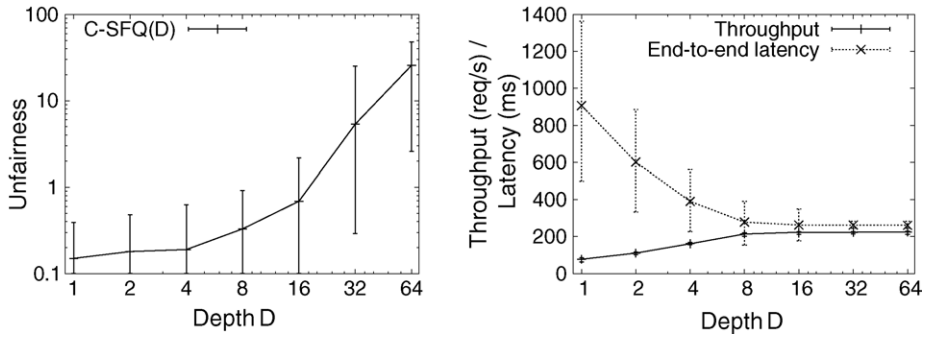

Fig. 6. Analysis of the effects of the degree of concurrency D to C-SFQ(D). The graph on the left shows the relation between unfairness and values of D. The graph on the right shows the aggregate throughput and the end-to-end latency obtained from the service for different values of D. The dots are the median values and the error bars show the 5th and the 95th percentile of the measurements.

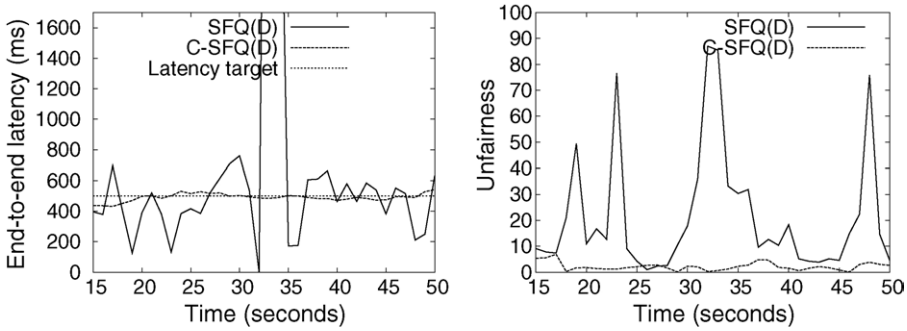

Fig. 7. Demonstrating the ability of SFQ(*D*) and C-SFQ(*D*) to meet performance goals when the STR controller from Section 2.2 sets the weights every 1 s. The two graphs show latency and unfairness.

5.4. SFQ(D) and C-SFQ(D) with an adaptive controller

This section demonstrates that the high unfairness of SFQ(*D*), when weights vary, impairs the feedback loop's ability to control the system in practice. We compare it against C-SFQ(*D*), in a closed loop system that uses the controller from Section 2.2 to automatically adjust weights according to system and workload dynamics.

Fig. 7 plots the end-to-end latency and unfairness for one of four flows over a 35 s time window of the execution. The results for the other flows are similar; they are omitted to avoid overcrowding the graphs. There are two main observations to make. First, the actual performance is close to the latency goal with C-SFQ(*D*). (The small violations of the goals are due to the nature of the controller. Typically, a controller takes no corrective action unless there is a violation of the goal.) The performance of SFQ(*D*), on the other hand, fluctuates much more widely. There are times at which it is completely off the goal. Second, during the periods in which the performance is far from the goal for SFQ(*D*), the unfairness is high. The unfairness peaks at 18, 23 and especially between 30 and 35 s. This shows that fairness is crucial for successfully controlling the system to achieve performance goals.

6. Related work

The possibility of using feedback from the system to dynamically control a scheduling mechanism and meet performance goals has been discussed in the literature [9,10]. In one case, flows are assigned fixed reservations, which are enforced using resource-specific schedulers for CPU and disk. Feedback about the actual performance each flow receives is used to decide how to share any unused resource capacity among active flows [9]. In another case, a learning heuristic is used to adjust flow reservations, so as to maximize the number of flows that meet their response latency goals [10]. Existing research has neither identified the desirable properties of schedulers to be used with such feedback loops, nor analyzed the effectiveness of existing schedulers in that context. Our work is applicable to the design of any such feedback-based resource control approach.

Extensive research in scheduling for packet switching networks has yielded a group of Weighted Fair Queuing variants for link sharing in communication networks, including WFQ [20], SCFQ [17], and SFQ [18]. Fair queuing has been adapted to other contexts such as disk scheduling [6], CPU scheduling [21], and server resource management [7].

7. Conclusions

In this paper, we are concerned with the problem of enforcing application-level performance goals in shared computing infrastructures. We proposed doing this by varying the parameters of proportional share schedulers using adaptive feedback-based control. Proportional share schedulers are most commonly implemented using variants of Weighted Fair Queuing (WFQ). We proved that existing WFQ schedulers are unfair and unpredictable when the flow weights vary. That makes them ineffective in the presence of dynamic control. We defined controllable fairness, a stronger notion of fairness for this case, we proposed a tag adjustment algorithm that ensures that WFQ schedulers are controllable-fair, and proved the properties of the resulting schedulers.

To validate the analytical results, we performed an experimental evaluation using a three-tier Web service. We confirmed that a typical concurrency-controlled WFQ scheduler, SFQ(D), exhibits poor fairness when flow weights vary by as little as 1%. On the other hand, a controllable-fair WFQ variant, C-SFQ(D), is shown to exhibit fairness that is in average two orders of magnitude better. We perform a sensitivity analysis of the controllable fairness of C-SFQ(D) against the values and deltas of flow weights and the degree of concurrency, which shows that C-SFQ(D) can be used even with aggressive controllers. Finally, we demonstrate that, due to its good controllable fairness, C-SFQ(D) can indeed be used with an adaptive feedback controller to enforce performance goals in shared services.

References

[1] A. Kamra, V. Misra, E. Nahum, Yaksha: a self-tuning controller for managing the performance of three-tiered web sites, in: Proceedings of the International Workshop on Quality of Service (IWQoS), Montreal, Canada, 2004, pp. 47–56.
[2] M. Karlsson, C. Karamanolis, X. Zhu, Triage: performance isolation and differentiation for storage systems, in: Proceedings of the International Workshop on Quality of Service (IWQoS), Montreal, Canada, 2004, pp. 67–74.
[3] P. Goyal, X. Guo, H.M. Vin, A hierarchical CPU scheduler for multimedia operating systems, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, 1996, pp. 107–121.
[4] W. Jin, J. Chase, J. Kaur, Interposed proportional sharing for a storage service utility, in: Proceedings of the International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS), New York, NY, USA, 2004, pp. 37–48.
[5] M. Karlsson, X. Zhu, C. Karamanolis, An adaptive optimal controller for non-intrusive performance differentiation in computing services, in: Proceedings of the IEEE Conference on Control and Automation (ICCA), Budapest, Hungary, 2005.
[6] P. Shenoy, H. Vin, Cello: a disk scheduling framework for next generation operating systems, in: Proceedings of the International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS), Madison, WI, 1998, pp. 44–55.
[7] B. Urgaonkar, P. Shenoy, T. Roscoe, Resource overbooking and application profiling in shared hosting platforms, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, 2002, pp. 239–254.
[8] T. Abdelzaher, K.G. Shin, N. Bhatti, User-level QoS-adaptive resource management in server end-systems, IEEE Trans. Comput. 52 (5) (2003).
[9] M. Aron, Differentiated and Predictable Quality of Service in Web Server Systems, Ph.D. Thesis, Computer Science Department, Rice University, 2000.
[10] V. Sundaram, P. Shenoy, A practical learning-based approach for dynamic storage bandwidth allocation, in: Proceedings of the International Workshop on Quality of Service (IWQoS), Monterey, CA, 2003, pp. 479–497.
[11] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, T. Lee, Performance virtualization for large-scale storage systems, in: Proceedings of the Symposium on Reliable Distributed Systems (SRDS), Florence, Italy, 2003, pp. 109–118.

[12] J. Chase, D. Anderson, P. Thakar, A. Vahdat, R. Doyle, Managing energy and server resources in hosting centres, in: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), Banff, Canada, 2001, pp. 103–116.
[13] K. Shen, H. Tang, T. Yang, L. Chu, Integrated resource management for cluster-based internet services, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, 2002, pp. 225–238.
[14] Y. Lu, T. Abdelzaher, C. Lu, G. Tao, An adaptive control framework for QoS guarantees and its application to differentiated caching services, in: Proceedings of the International Workshop on Quality of Service (IWQoS), Miami Beach, FL, 2002, pp. 23–32.
[15] K.J. Åström, B. Wittenmark, Adaptive control, Electrical Engineering: Control Engineering, 2nd ed., Addison-Wesley Publishing Company, 1995, ISBN 0-201-55866-1.
[16] C. Karamanolis, M. Karlsson, X. Zhu, Designing controllable computer systems, in: Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS), Santa Fe, NM, 2005.
[17] J. Davin, A. Heybey, A simulation study of fair queuing and policy enforcement, Comput. Commun. Rev. 20 5 (1990) 23–29.
[18] P. Goyal, H.M. Vin, H. Cheng, Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks, IEEE/ACM Trans. Networks 5 (5) (1997) 690–704.
[19] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, J. Chase, Correlating instrumentation data to systems states: a building block for automated diagnosis and control, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, CA, 2004, pp. 231–244.
[20] A. Demers, S. Keshav, S. Shenker, Analysis and simulation of a fair queuing algorithm, in: Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM), Austin, TX, 1989, pp. 1–12.
[21] P. Goyal, X. Guo, H. Vin, A hierarchical cpu scheduler for multimedia operating systems, in: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, 1996, pp. 107–121.

Magnus Karlsson is a senior researcher at HP Labs, Palo Alto, California. His research interests include general design methods for the management of computer systems, adaptive and nonlinear control theory, estimation theory, and QoS. Magnus received his PhD in Computer Engineering at Chalmers University of Technology in Gothenburg, Sweden in 1999.

Christos Karamanolis is a senior researcher at HP Labs, Palo Alto, California. His research interests include the design and management of distributed systems and in particular enterprise storage systems. He received a Diploma in Computer Engineering from the University of Patras, Greece and a PhD in Distributed Computing from Imperial College, University of London, UK.

Jeffrey S. Chase is an Associate Professor of Computer Science at Duke University in Durham, NC. His research with Duke's Internet Systems and Storage Group deals with efficient and reliable sharing of information and resources in computer networks ranging from clusters to the global Internet. Dr. Chase is an alumnus of Dartmouth College. He received his PhD in Computer Science from the University of Washington in 1995.