Controllable Fair Queuing for Service Utilities

Magnus Karlsson, Christos Karamanolis, Jeff Chase[1]
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2005-21
February 7, 2005*

E-mail: karlsson@hpl.hp.com, christos@hpl.hp.com, chase@cs.duke.edu

fair queuing, closed-loop systems, utility computing

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 schedule when the weights change, and proves that the algorithm results in controllable-fair WFQ 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 depth-controlled Start-time Fair Queuing, which supports interposed proportional sharing for complex network services with no internal support for differentiated service quality. The prototype includes an adaptive control system that adjusts the flow weights on the C-SFQ(D) scheduler to meet latency and throughput targets under a variety of conditions. Experimental results demonstrate the importance of controllable-fair scheduling for feedback control of network services.

* Internal Accession Date Only
[1]Duke University, Durham, North Carolina, USA

Approved for External Publication

Controllable Fair Queuing for Service Utilities

Blind Review

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* schedule when the weights change, and proves that the algorithm results in controllable-fair *WFQ* 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 depth-controlled Start-time Fair Queuing, which supports interposed proportional sharing for complex network services with no internal support for differentiated service quality. The prototype includes an adaptive control system that adjusts the flow weights on the *C-SFQ(D)* scheduler to meet latency and throughput targets under a variety of conditions. Experimental results demonstrate the importance of controllable-fair scheduling for feedback control of network services.

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; this can improve resource efficiency while protecting against unplanned demands and service outages.

Shared service utilities must manage their physical resources in a way that meets the performance needs of the customers and their workloads. Typically, negotiated *Service Level Agreements* (SLAs) define contractual performance objectives—response time bounds and minimum throughput requirements—for the requests of different customers or workloads. Without loss of generality, we suppose that SLAs are defined at the granularity of *flows* of requests arriving at each network service. The service handles the requests of all flows using shared physical resources such as CPUs, memory, network switches, and disk spindles. Utility resource control apportions resources to meet the demands of each service or flow. To adapt to changing conditions, resource managers may monitor resource status and flow performance metrics to adjust performance under feedback control [1, 23]. The actuators available to the utility controller include admission control, throttling admitted request flows or changing the scheduling of request execution [17, 21, 16, 24], and/or changing the assignment of resources such as disks [2].

This paper focuses on performance control by varying the shares of resources available to each request flow. In particular, we focus on controllability properties of proportional share schedulers, which are most commonly implemented using variants of weighted fair queuing (*WFQ*). The input to a proportional share scheduler is a vector of *weights*, one for each flow; the scheduler partitions resources or service capacity among the flows in proportion to their weights. A feedback controller can vary the weights dynamically to enforce its performance objectives. This general approach is based on the simple premise that performance of a workload varies in a predictable way with the amount of resource available to execute it. This approach may be used in tandem with other actuators to account for the diversity of factors that affect workload performance. A few recent research projects have explored the feasibility of controlled resource sharing driven by feedback from workload performance measures [29, 25, 23, 3, 1, 21, 24, 22].

A key problem for performance control with *WFQ* schedulers is that dynamic changes to the flow weights take effect only after a variable time lag. This effect results from the way that *WFQ* schedulers implement their *work-conserving* property: if idle resources exist, then flows may exceed their allotted shares to consume them without penalty. Schedulers with this property use resources efficiently to deliver better performance for a given weight vector in the presence of transient load changes, freeing the control system from the need to adjust the weights in response to transient changes. Unfortunately, the implementation of this property in *WFQ* schedulers makes them less responsive to changes in the weights, thus impeding stable feedback control. In a formal sense, their fairness properties are 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 *con-*

trollable fairness required to allow stable control, develops a tag adjustment 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 controllable request scheduler called *C-SFQ(D)*. *C-SFQ(D)* is a controllable-fair variant of an *interposed* request scheduler [21] for complex network services. The scheduler is placed on the network path between the service and its clients; it intercepts requests sent to the service and re-orders or delays them to enforce approximate proportional sharing of the service's capacity to serve requests (as shown in Figure 1). An interposed scheduler releases up to a depth D of requests for concurrent processing by the service; we show that the depth D must also be subject to dynamic control, and show how *C-SFQ(D)* preserves controllable fairness across changes to D.

We conducted experiments using *C-SFQ(D)* and a feedback control system to meet performance objectives for a a 3-tier Web application service. The results show that *C-SFQ(D)* can be used with adaptive setting of weights and D to effectively enforce throughput and latency goals in the 3-tier system. The simple control system used in the experiments is stable when used in conjunction with a controllable-fair scheduler, but we show that a conventional *WFQ* scheduler that is not controllable-fair causes the control system to become unstable. A comprehensive study of effective control systems for complex services with controllable-fair schedulers is a topic for future work.

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). Examples of utility service providers include shared storage services [27, 2, 25, 9, 24, 21, 23] or shared clusters hosting a multi-tier Internet application for each customer [10, 26]. The objective of the service provider is to meet the performance targets specified in the SLAs in order to maximize yield or minimize penalty. At the same time, the service provider seeks to minimize costs by using its resources efficiently.

The clients accessing the service generate streams of *requests* that may use hardware resources—processor capacity, disk arms, I/O bandwidth—in widely varying ways. Requests are grouped into service classes called *flows*. For example, all the clients accessing a given storage volume or application service could be grouped into the same flow, or different clients of a given service may be grouped into different flows to allow for differentiated service quality. The objective is to meet the performance targets of the request flows by controlling the rates at which they consume the service resources.

2.1 Resource Control

This paper deals with utility schedulers that adjust the performance of each request flow by controlling the resources available to execute it. A scheduler limits resource consumption of each flow f in proportion to a *weight* ϕ_f assigned to each flow. If flow weights are normalized to sum to one, then we may interpret each weight 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, because the heavily loaded flow is limited to its configured share of resources.

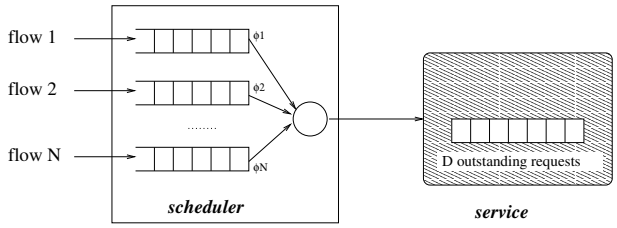

Figure 1: A request scheduler is interposed on the network path between a service and its clients. The scheduler intercepts and delays and/or re-orders the requests to meet performance goals defined per flow, e.g., by allocating resources in proportion to a share or weight ϕ associated with each flow. The interposed scheduler views the service as a black box, and enforces a maximum degree of concurrency D in the service.

Of course, a complex set of factors determines the application performance for a given weight setting. If the weight represents a share of the system's capacity to serve requests, then this approach enforces throughput targets directly. It also supports soft response time targets: for a configured request completion rate, the distribution of the flow's request arrivals and service demands determines the response time profile. Statistical delay guarantees can be proven under various assumptions about arrival processes, service demands, and allocation of surplus resources [32].

We focus specifically on *work-conserving* proportional-share schedulers based on the widely used technique of *weighted fair queuing* (*WFQ*)—see Section 2.3. In contrast to guaranteed reservations, which guarantee 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* or *backlogged*, i.e., they have backlogs of queued requests. Work-conserving schedulers use resources more efficiently, and they improve performance of active flows when the system is lightly loaded. These properties limit the need for fine-grained adjustments in the weights to accommodate bursty demand.

This paper focuses on the use of *WFQ* schedulers in conjunction with dynamic control loops to adjust the flow weights. The controller may continuously adjust the weights in response to observable performance metrics (response latency and throughput) obtained by each flow. The use of dynamic control to meet performance goals has been discussed in the literature [3, 10, 29, 1, 23]. Indeed, we expect that a number of different implementations of a feedback loop will be possible. The actual implementation will depend on the dynamics of the target system and the time granularity at which actions must be taken. At a coarse granularity, a system administrator might configure the scheduler manually, according to currently observed performance or in anticipation of future activities. Alternatively, an automated tool may use a learning heuristic [29] or an adaptive controller [23] to adjust automatically to changes in the system.

Section 5.4 presents experimental results with a prototype control system for a 3-tier Web service. However, this paper focuses on the properties of *WFQ* proportional-share schedulers that are important for stable feedback control, rather than on the design and analysis of the control system itself.

2.2 Controllable Schedulers

Whatever the design of the feedback loop, the control system must adjust the scheduler parameters dynamically. However, *WFQ* scheduling algorithms have been designed and analyzed for weights that are fixed over time. In particular, these algorithms schedule requests based on tags that are assigned at request arrival time, due to the way that they maintain fairness and avoid penalizing flows that consume idle resources. As a result, changes to the weights do not take effect until queued requests have exited the system. In the worst case, incorporating *WFQ* schedulers into feedback loops can produce unbounded delays in reaction time, and violate scheduler fairness properties arbitrarily, as shown in Section 3. While a less aggressive control loop can tolerate longer reaction times, variable reaction times may cause the control loop to become unstable. Bursty demands can cause large variations in the queued lengths, and thus reaction times that vary in a fashion that is unpredictable to the controller.

We define three properties that are important for a *controllable* scheduler that can be used effectively in a dynamic control loop to meet application-level performance goals:

- *Predictability*. Ideally, there is a monotonic relation between a flow's weight and its performance at a given load level. For example, increasing the share of a flow may reduce the average latency of that flow's requests, but should never increase it. This monotonic relation is necessary in any closed-loop to estimate a model of the performance of the system as a function of share settings. Note that a work-conserving scheduler may exhibit transient improvements in a flow's performance when the system has surplus resources, independent of the flow's weight. What is important is that performance is stable and predictable for a given workload.
- *Fairness*. The partitioning of service capacity should resemble the specified proportional shares of the flows that compete for the service. In a complex system, artifacts such as the assignment of data to disks or servers may affect the performance from a given set of weights, but the scheduler's fairness must be consistent across changes to the weights. This is a key requirement for predictability.
- *Responsiveness*. A controllable scheduler must have a *known reaction delay*. That is, there is a known time lag between changing some parameters (e.g., flow weights) and observing the effects of that change. This property is necessary to ensure a stable closed-loop system. If the effects of a change are not stable and observable within the control interval, then the controller may overreact and amplify its earlier actions. This may give rise to oscillations and instability.

In this paper, we use both analytical and experimental results to show that existing *WFQ* schedulers are *not* controllable in a well-defined sense, and that control loops using uncontrollable schedulers can become unstable.

2.3 Weighted Fair Queuing

Many variants of *WFQ* scheduling algorithms have been developed and extensively studied in the literature (see Section 6). All *WFQ* variants are designed following the same principles. Each *flow* f consists of a sequence of requests $p_f^0 \ldots p_f^n$ arriving at the server. Each request p_f^i has an associated cost c_f^i bounded by a constant c_f^{max}. For example, the requests may be packets of varying lengths or requests of varying costs. Fair queuing allocates the capacity of the resource in proportion to *weights* assigned to the competing flows. The weights may represent service rates, such as bits or cycles or requests per second. 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 service capacity, and that request costs are normalized to a service capacity of one unit of cost per unit of time.

WFQ schedulers are *fair* in the sense that active flows share the available service capacity proportionally to their weights, within some tolerance that is bounded by a constant over any time interval. Formally, if $W_f(t_1,t_2)$ is the aggregate cost of the requests from flow f served in any time interval $[t_1,t_2)$, then a fair scheduler guarantees that:

$$\left| \frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g} \right| \leq U_{f,g} \tag{1}$$

where f and g are any two flows continuously backlogged with requests during $[t_1,t_2)$ and $U_{f,g}$ is a constant that depends on the flow weights and the maximum cost of flow requests. All algorithms try to ensure low values of $U_{f,g}$, which indicates better fairness. Table Table 1 summarizes the symbols used in this paper.

WFQ schedulers dispatch requests in order of *tags* assigned at request arrival time. When the j^{th} request 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:

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

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

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

where $A(p_f^j)$ is the arrival time of request p_f^j, c_f^j is the actual cost for the service to execute request p_f^j and ϕ_f is the weight of flow f.

These tags represents the time at which each request 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: all flows are backlogged, the server completes work at a fixed ideal rate, request costs are accurate, and the weights sum to the service capacity. In an idealized bit-by-bit fair round-robin scheduling algorithm, each active flow f receives ϕ_f bits of service per unit of virtual time: $v(t)$ speeds up when surplus resources are available to serve active flows at a higher rate. Calculating $v(t)$ exactly is computationally expensive; in particular, the cost is prohibitive when the capacity of the service fluctuates [15]. *WFQ* algorithms differ primarily in the way that they approximate virtual time.

There are two *WFQ* algorithms that approximate $v(t)$ efficiently by clocking the rate at which the service actually completes work. Self Clocked Fair Queuing (*SCFQ*) [12, 15] and Start-time Fair Queuing [17] (*SFQ*) approximate $v(t)$ with (respectively) the finish tag or start tag of the request in service at time t. The main advantage of *SFQ* over *SCFQ* is that it reduces the maximum delay incurred for the processing of individual requests, by scheduling requests in increasing order of start tags [17]. For simplicity, our analysis of controllability for *WFQ* algorithms focuses on self-clocking algorithms, but the proofs are independent of how $v(t)$ is computed.

2.4 Interposed Request Scheduling

Our experimental work focuses on resource control solutions that consider the service as a "black box". Papers in last year's SIGMETRICS show that proportional-share schedulers can meet SLA objectives for complex services such as network storage by interposing a scheduler that controls the dispatch of requests into the service [21, 19], as depicted in Figure 1. Other systems have used a similar idea [25, 9, 21, 23, 22]. The main advantage of this approach is that it is non-intrusive and general; it treats the service as a "black box" and is applicable even to services that have no internal support for differentiated service quality. It assumes that the individual requests in each flow have modest service demands (costs) that are known approximately: there is a reasonably high volume of requests per flow with reasonably stable average case behavior.

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 so are not suitable for interposed request scheduling. A *depth-controlled* variant of *SFQ* (*SFQ(D)*) with proven fairness bounds has been developed for interposed request scheduling for concurrent requests [21]. The maximum concurrency D reflects a trade-off between utilization of the service and the worst-case fairness bound of the scheduler. In this case, $U_{f,g}$ also depends on D.

Section 4 proposes a tag adjustment algorithm to ensure that *WFQ* schedulers are controllable. We develop a controllable variant of *SFQ(D)* called *C-SFQ(D)*, and we present experimental results evaluating the *C-SFQ(D)* scheduler in conjunction with a control system using interposed request scheduling for a 3-tier Web service. An additional factor for controllable interposed request scheduling is that the depth parameter D is also subject to adjustment by the resource controller. In particular, higher concurrency (higher values of D) may improve throughput, and it may be necessary to increase D if hotspots develop within the service; however, higher values of D may impede the system's ability to meet tight response time bounds. At the same time, dispatched requests are not responsive to changes in the weights, so higher values of D makes the scheduler less responsive.

3. WFQ IS NOT CONTROLLABLE

All *WFQ* algorithms have well-defined fairness bounds. The best known *WFQ* algorithms, including *WF*^{2}*Q* and the self-clocking algorithms *SCFQ* and *SFQ*, have been shown to have a fairness bound that is:

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

which is two times the theoretical lower upper bound for any fair queuing algorithm [15].

The proven fairness bounds for *WFQ* schedulers assume that flow weights are fixed. We show in this section that the bounds do *not* hold when weights change dynamically. In fact, we prove that *WFQ* algorithms cannot ensure any fairness bound in the general case under dynamic control of the weights. For the proofs in this section, we refer to *WFQ* algorithms that emulate $v(t)$ by the start tag of the last submitted request (e.g., *SFQ* [17]). The proofs are similar for algorithms that emulate $v(t)$ by the finish tag of the last submitted request (e.g., *SCFQ* [12]), but are omitted due to lack of space. We assume without loss of generality that all requests have unit cost.

Symbol	Meaning
$\phi_f(i)$	Weight of flow f during time interval i.
p_f^j	The j-th request of flow f.
c_f^j	Cost of request p_f^j.
$c_f^{max}(i)$	Maximum cost for a request from flow f during time interval i.
$A(p_f^j)$	Arrival time of request p_f^j.
$S(p_f^j)$	Start tag of request p_f^j.
$F(p_f^j)$	Finnish tag of request p_f^j.
$v(t)$	Virtual time at time t.
$W_f(i)$	Total amount of work/cost served from flow f during time interval i.
$D(i)$	The maximum number of outstanding requests during time interval i.
$D'(i)$	The actual number of outstanding requests 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.

Table 1: Frequently used symbols in this paper.

We first define the notion of an *interval* in the presence of changing scheduler parameters:

DEFINITION 1. *A time interval i is a period of time $[t_i, t_{i+1})$ during which $\phi_f(i)$ is held constant for every flow f.*

When flow weights can change dynamically, i.e., $\phi_f(i) \neq \phi_f(i-1)$ for flow f, then according to the following theorem, there exist intervals in which a flow may receive no service irrespectively of its weight setting.

THEOREM 1. *There exists an interval i during which flows f and g are continuously backlogged, $\phi_f(i) \neq 0$ and $\phi_g(i) \neq 0$, but the bound (5) does not hold for that interval.*

Proof: Consider the example of Figure 2 with two continuously backlogged flows f and g. Suppose during time interval 1, $\phi_f = 0.01$ and $\phi_g = 0.99$. f has one request served, and the start tag of the next request 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 requests 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 request from f is processed, as $S(p_f^2) = 100$, well ahead of $v(t)$. In fact, $\phi_f = \infty$ would produce exactly the same result, as the start tag of the second request of f was computed using the weight during interval 1. This counter example shows that there exist intervals during which the bound of (5) does not hold. □

THEOREM 2. *For any time interval i during which flows f and g are backlogged during the entire interval, the fairness of a WFQ algorithm during i is bounded only by:*

$$U_{f,g}(i) = \max\left(\frac{W_f(i)}{\phi_f(i)}, \frac{W_g(i)}{\phi_g(i)}\right) \quad (6)$$

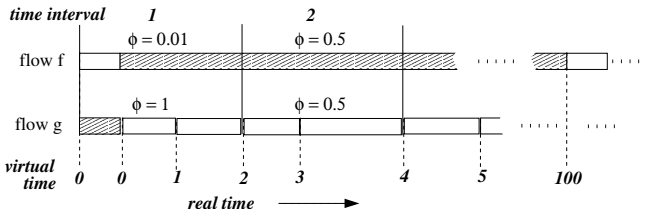

Figure 2: Example showing that, when weights change in *SFQ*, there exist intervals in which a flow receives no service independent of its weight setting. Flows f and g are continuously backlogged. The white blocks depict request execution; the gray blocks depict backlog but no execution.

Proof: Using $|A-B| \leq \max(A,B)$ and equation (1), we can trivially see that (6) is the worst upper bound for fairness during interval i. From the counter example in the proof of Theorem 1, we can see that this indeed defines the lag in work served for f and g in a worst-case scenario. Thus there are intervals during which the fairness is bounded only by the work arriving for flow g, i.e., by $U_{f,g} = \frac{W_g(i)}{\phi_g(i)}$ and $W_f(i) = 0$. □

This is not a good bound, because it is not a function of request costs (which are known constants), but of the service received by some flow, which can be arbitrarily high depending of weight settings in previous intervals. Thus the scheduler has no bound on its reaction delay. To illustrate this with an example, consider again the scenario in Figure 2. During interval 2, $W_f(2) = 0$, which makes the bound $U_{f,g}(2) = \frac{W_g(i)}{\phi_g(i)}$, i.e. it increases with the amount of work flow g completes during this interval. Only at a later point, when $v(t) = 100$, will f eventually have its next request served.

In Section 5.4 we demonstrate empirically that, as a result of this variable and unbounded reaction time, a *WFQ* scheduler (*SFQ(D)* in this case) cannot be used effectively by a controller that sets weight values dynamically to meet performance goals. The system exhibits wide oscillations and instabilities. An unmodified original *WFQ* scheduler cannot effectively enforce differentiation when weights change. To capture the fairness of a scheduler when weights vary dynamically, we introduce the notion of *controllable fairness*.

DEFINITION 2. *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 $\phi_f(j)$ and $\phi_g(j)$ are constant within each interval i, $i \in T$, controllable fairness is defined as:*

$$\sum_{i \in T} \left| \frac{W_f(j)}{\phi_f(j)} - \frac{W_g(j)}{\phi_g(j)} \right| \leq U^*_{f,g} \tag{7}$$

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

In order to improve the poor fairness bound of *WFQ* schedulers, we need to recalculate the tags of backlogged requests when weights change. One naive way of doing this would be to use equations (2) - (4) to recompute the tags of all backlogged requests, every time the weights change. To avoid the problem identified in Theorem 2, we must ignore the finish tags of requests submitted in previous intervals—all flows start with a clean slate ($F(p^0_f) = 0$) for this interval. This re-computation would indeed result in a good fairness bound for every *single* interval i, as given by equation (5). 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*.

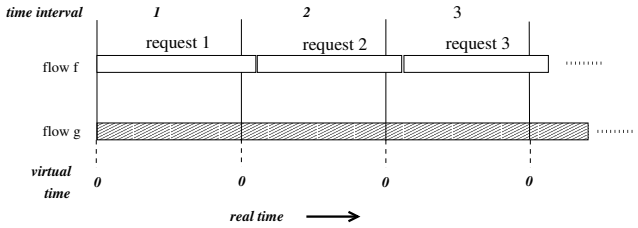

Figure 3: Example showing that it is possible to construct an unbounded number of consecutive intervals during which there is a flow that receives no service, even though it has non-zero weights.

THEOREM 3. *When the flow weights can vary and the tags of backlogged requests are recomputed using equations (2) - (4) every time some weights change, then the controllable fairness is unbounded.*

$$U^*_{f,g} = \infty \tag{8}$$

Proof: We use a counter-example to show that $U^*_{f,g}$ is unbounded as $N \to \infty$. Consider the scenario of Figure 3. There are two flows f and g that are continuously backlogged during an infinite sequence of intervals. The start tags of the first requests of both flows are set to 0 at the beginning of interval 1. *WFQ* arbitrarily picks to submit the request 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 request of flow f is still not completed. Thus, virtual time is still 0. After recomputing the tags of the backlogged requests, the start tags of the next requests of both flows are again 0. The outstanding request from f completes during interval 2 and *WFQ* arbitrarily picks to submit the next request 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 service, while g gets nothing. Of course, such worst-case scenarios are *not* restricted to cases where the duration of an interval is shorter than that of a request execution. It is easy to construct similar worst-case scenarios for the case where the weight of a flow is less than 100% of the total weight but sufficient to use up entirely the service for the duration of an interval. Theorem 3 follows directly from the above counter-example. □

The intuition behind Theorem 3 is that within each interval fairness is bounded by (5). However, it is possible to reach this bound at every single interval as the re-computation phase starts *WFQ* from scratch each time. This means that the bounds add up and thus as the length of the sequence of intervals reaches infinite ($N \to \infty$), the aggregate bound is also infinite. 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 services that process one request at a time. The results are trivially applicable to depth-controlled *WFQ* variants (e.g., *SFQ(D)* [21]) and services that support higher degrees of concurrency, $D \geq 1$.

4. CONTROLLABLE WFQ

The problem with the naive *WFQ* extension discussed in the previous section is that it does not account for service capacity used up by requests submitted in previous intervals. All tags are recomputed from scratch. As a result, there are cases where some flows may use the service exclusively for an infinite number of intervals starving some other flows. In this section we propose an extension to *WFQ* algorithms that provably provides good controllable fairness and thus good predictability and responsiveness when the flow weights change. Again, the discussion here focuses on *WFQ* algorithms that emulate $v(t)$ by the start tag of the last submitted request. In particular, we present and analyze an algorithm called *Controllable SFQ*, *C-SFQ* for short, which is an extension of *SFQ*. However, 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*.

$$F(p_f^{j-1}) = S(p_f^{j-1}) + \frac{c_f^{j-1}}{\phi_f(i)} \quad (9)$$

$$S(p_f^k) = \max(v(t), F(p_f^{k-1})),\ j \le k < j+Q_f \quad (10)$$

$$F(p_f^k) = S(p_f^k) + \frac{c_f^k}{\phi_f(i)},\ j \le k < j+Q_f \quad (11)$$

Equation (9) recomputes the finish tag of the last submitted request of flow f (in some interval before i), as if it have had the new weight setting. The tags of the backlogged requests are adjusted accordingly in equations (10) and (11) which are equivalent to (3) and (4) of *WFQ*. Re-computation (9) 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 fairness-bound theorem for *C-SFQ* is that *C-SFQ* behaves exactly like *SFQ* within each interval (the virtual clocks of the flows are reset only at the beginning of an interval). Thus, the fairness bound within every single interval is the same as that of *SFQ*.

THEOREM 4. *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:*

$$U^*_{f,g} = \max_{i \in T} \left(\frac{c_f^{max}(i)}{\phi_f(i)} + \frac{c_g^{max}(i)}{\phi_g(i)} \right) \quad (12)$$

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 interval 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 *C-SFQ* 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 those received by *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 (10) 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 $U_{f,g} = \left(\frac{c_f^{max}}{\phi_f} + \frac{c_g^{max}}{\phi_g} \right)$ [17]. 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:

$$U^*_{f,g}(i) = \left(\frac{c_f^{max}(i)}{\phi_f(i)} + \frac{c_g^{max}(i)}{\phi_g(i)} \right) \quad (13)$$

Thus, the fairness bound across a sequence of intervals is the worst bound among all individual intervals in the sequence, given by equation (12). □

Since support for high degree of concurrency is important in computing services, we discuss here a depth-controlled *WFQ* variant. In particular, we present and analyze an algorithm called *Controllable SFQ(D)*, *C-SFQ(D)* for short, which is an extension of depth-controlled Start-tag Fair Queuing *SFQ(D)* [21]. As shown in Section 5.2, the maximum depth D is a scheduler parameter that also needs to be adjusted, along with flow weights, according to system and workload dynamics. A controllable scheduler must be fair even when D changes. The original fairness bound for *SFQ(D)* for when

weights and D do not change is [21]:

$$U_{f,g} = (D+1)\left(\frac{c_f^{max}}{\phi_f} + \frac{c_g^{max}}{\phi_g}\right) \quad (14)$$

Theorem 5 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:*

$$D'_{max}(i) = \max(D(i), D(j)) \quad (15)$$

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 $j \leq k < i$. There are two cases to consider for interval i:

1) If $D'(k) \leq D(i)$, 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 $D(m)$, $m < j$. □

The following theorem applies to any sequence of intervals, even sequences of infinite length.

THEOREM 5. *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:*

$$U^*_{f,g} = \max_{i \in T}\left((D'_{max}(i)+1)\left(\frac{c_f^{max}(i)}{\phi_f(i)} + \frac{c_g^{max}(i)}{\phi_g(i)}\right)\right) \quad (16)$$

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

Proof: When the depth is changed between intervals, the maximum possible number of pending requests during some interval i is given by $D'_{max}(i)$ in equation (15). According to (14), the bound for a specific interval i is then $(D'_{max}(i)+1)(\frac{c_f^{max}(i)}{\phi_f(i)} + \frac{c_g^{max}(i)}{\phi_g(i)})$. Thus, the worst-case bound in sequence T is the highest bound of any single interval $i \in T$, as given by equation (16). □

In *C-SFQ(D)*, we now have a scheduler that is controllable, i.e., it provably satisfies all the requirements stipulated in Section 2.1. In the next section, we will show how *C-SFQ(D)* can be used together with a feedback loop to achieve performance goals for a real system, and how previous non-controllable schedulers cannot be used in this setting.

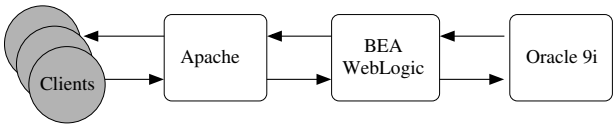

Figure 4: The 3-tier system that is used for the experimental evaluation.

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. The system consists of three components: a web server, an application server and a database server. As depicted in Figure 4, client requests arrive at the web server and, unless they are for static content, 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.

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 3-tier system is a version of the Java PetStore [20] 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 [11], e.g., browsing, searching and purchasing behaviors including the corresponding time scales and probabilities these occur with. The workload is generated by `httperf` on a separate machine that is identical to the ones above but runs Linux. For the experiments in the rest of this section, we generate 75 concurrent client sessions unless otherwise stated. For these experiments, we usually consider two flows f and g. Flow f consists of 38 clients and flow g of 37. The sample interval for gathering statistics and for changing the weights and D is always set to 3 seconds.

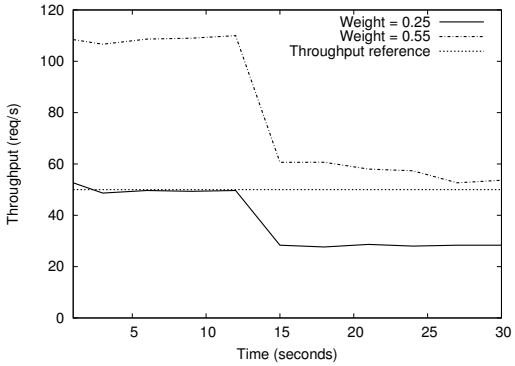

Figure 5: Demonstrates the need to rapidly adjust flow weights. The x-axis refers to the execution time of an actual run. The graph shows two runs of the same flow trace executing with weights of 0.25 and 0.55 respectively. The flow has a throughput goal of 50 req/s. Initially, a weight of 0.25 is sufficient for meeting the goal. At time 15 seconds, the clients of the flow shift from browsing-only to purchasing products. Now, the weight needs to be 0.55 to meet the goal.

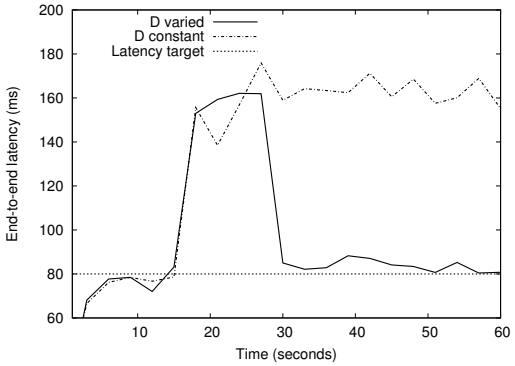

Figure 6: Demonstrating the need to be able to adjust D. The x-axis refers to the execution time of an actual run. The graph shows two runs of the same flow trace. The flow has a latency goal of 80 ms. At time 15 seconds, the clients switch from browsing to purchasing products. At time 30 seconds in one run, D is adjusted from 32 to 8 to meet the latency goal.

5.2 Weights and D need to vary continuously

This section demonstrates that the weights and depth of a scheduler have to be continuously adjusted to meet performance goals. There are a number of reasons why the weights and D need to change, including:

- Variation to the service capacity. This may be due to changes in cache hit ratios, or variations in client behavior. For example, the transactions involved in product purchasing are more heavy-weight than simply browsing products.
- Changes to the number of clients accessing the service, possible due to a flash-crowd or due to diurnal patterns.
- Modifications to the service's hardware or software. For example, more CPUs are added, or a new version of some software is installed.

To see why weights need to be adjusted, consider the scenario of Figure 5. There are a number of flows accessing the system. One of these, flow f, has a throughput goal of 50 req/s which can initially be met with a weight setting $\phi_f = 0.25$. At time 15 seconds in the execution, the clients of this flow switch from from browsing-only to purchasing products. Browsing accesses mostly static content and thus consumes few service resource. Purchasing consumes more resources, as it involves more heavyweight transactions that access the database tier frequently. As a result, a weight of 0.25 is not be sufficient anymore for f to meet its throughput goal. Instead it would require a weight of 0.55, which is a relative weight change of of 120%. If a weight of 0.55 would have been chosen even when the clients were browsing products, the throughput goal would have been met by a wide margin, thus wasting valuable resources that other flows could have potentially used. It is thus desirable that the weights are adjusted dynamically in reaction to such workload changes.

Flow weights are not the only scheduler parameter that needs to be controlled. The maximum degree of concurrency D has also to be dynamically adjusted in the face of changes in the workloads or the system. We demonstrate this with the example of Figure 6. We have a flow with an end-to-end latency goal of 80 ms. At time 15 seconds, all clients of the flow switch from browsing to purchasing products. Up until time 15 seconds, the latency goal can be met with $D = 32$. After the workload change though, this D results in much higher response latencies. In one run, the value of D is adjusted down to 4, at time 30 seconds. This results in meeting the latency goal again.

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 depth. We use the following metric, called *unfairness* (E), to quantify effective differentiation.

DEFINITION 3. *For a set of flows F, the effectiveness by which the weight of flow $f \in F$ is enforced in the service during interval i is captured by*[1]*:*

$$E(f) = 100 \cdot \left| \frac{\phi_f(i)}{\sum_{g \in F} \phi_g(i)} - \frac{W_f(i)}{\sum_{g \in F} W_g(i)} \right| \qquad (17)$$

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. While the controllable fairness introduced is Section 3 refers to upper bounds for fairness, this metric captures the actual deviation from the specified flow weights during an execution. The higher the unfairness exhibited by a scheduler, the harder it is to control that scheduler for enforcing performance goals.

First, we quantify the unfairness of *SFQ(D)* and *C-SFQ(D)* by setting the weights in the same way as in the worst-case scenario of Figure 2. The interval duration is 3 seconds. Figure 8 shows the unfairness of flow f. With *SFQ(D)*, $E(f)$ is constantly at 50, after 10 seconds. This makes the scheduler uncontrollable, as we

[1]For the discussion in this paper, the denominator of the first fraction is always 1, as we assume weights are normalized to 1.

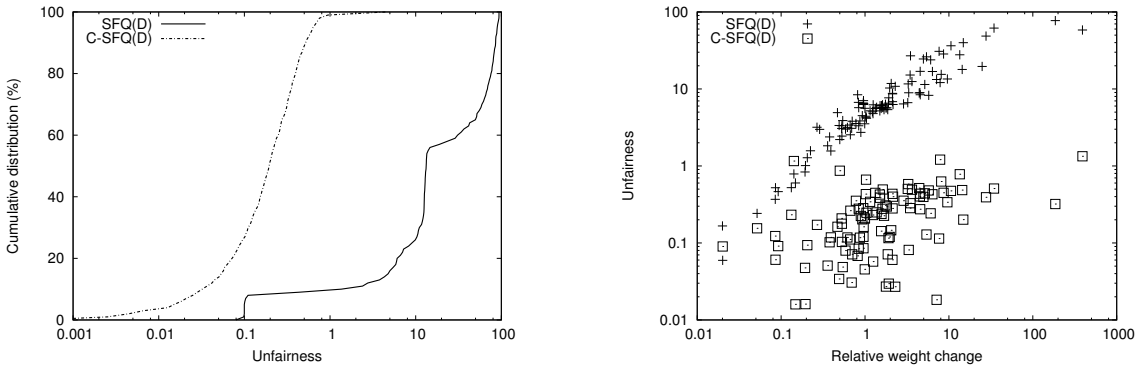

Figure 7: Sensitivity analysis of the unfairness of *SFQ(D)* and *C-SFQ(D)* against weight settings and changes. The graph on the left shows the CDF of unfairness for all the intervals of a run (the x-axis is in logarithmic scale). The graph on the right shows unfairness against the relative weight changes (both axes are in logarithmic scale).

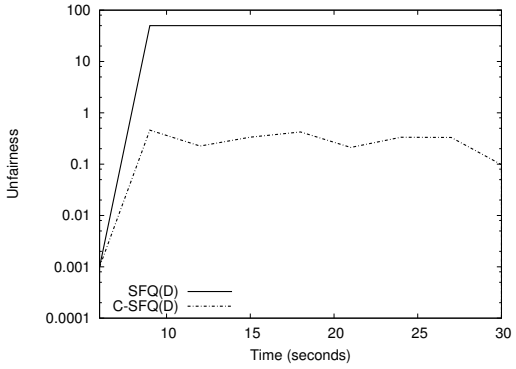

Figure 8: The unfairness of flow f with *SFQ(D)* and *C-SFQ(D)* in the three-tier system, when the weights are set according to the worst-case scenario of Figure 2. The x-axis refers to the execution time of an actual run.

demonstrate in Section 5.4. With *C-SFQ(D)*, on the other hand, $E(f)$ ranges between 0.1 and 0.5, which provides a system that can indeed be controlled.

Next, we analyze the effects of the *weight values* on unfairness. For the experiments reported in Figure 7, we consider runs with two flows, f and g. The flow weights are set randomly using a white noise generator. 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 $\phi_g = 100 - \phi_f$; $D = 4$. All 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 Figure 7 shows the cumulative distribution function of $E(f)$ for *SFQ(D)* and *C-SFQ(D)* across the 1,000 intervals of a run. From the graph, we can see 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. As was shown in Section 3, the reason for this high unfairness of *SFQ(D)* is that the tags of the pending requests are not changed when the weights change. We do not consider here the naive tag recalculation algorithm, due to the results of Theorem 3—it may result in arbitrarily bad fairness.

The example of Figure 2 showed the weight of a flow changing from 0.01 to 0.5, which is a change that is more than one order of magnitude in size. The larger the relative change, the longer it takes for weight settings to have any effect on the flows. To examine how a weight delta affects the unfairness of a scheduler, we have plotted the unfairness $E(f)$ of *SFQ(D)* and *C-SFQ(D)* as a function of relative weight change, in the right graph of Figure 7. Relative weight change in the case of two workloads f and g is defined as:

$$\phi_\Delta(i) = \frac{1}{2}\left(\frac{|\phi_f(i) - \phi_f(i-1)|}{\phi_f(i-1)} + \frac{|\phi_g(i) - \phi_g(i-1)|}{\phi_g(i-1)}\right) \quad (18)$$

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 feedback-based controllers.

Let us now examine how D, the maximum 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. Figure 9 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 5. 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, the degree of concurrency in the system 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 and, if possible, always have D outstanding requests. That means that differentiation is less effective than before, as weight settings have less impact on the performance of the system. Thus, work-conservation has a negative effect on the ability to control the system. As we will see in Section 5.4, the high unfairness levels of $D \geq 32$ indeed affect the ability of a controller to control the system, while the low levels of unfairness for $D \leq 16$ result in a controllable system.

On the other hand, work-conservation is a desired property as it increases the total throughput of the system and presumably the

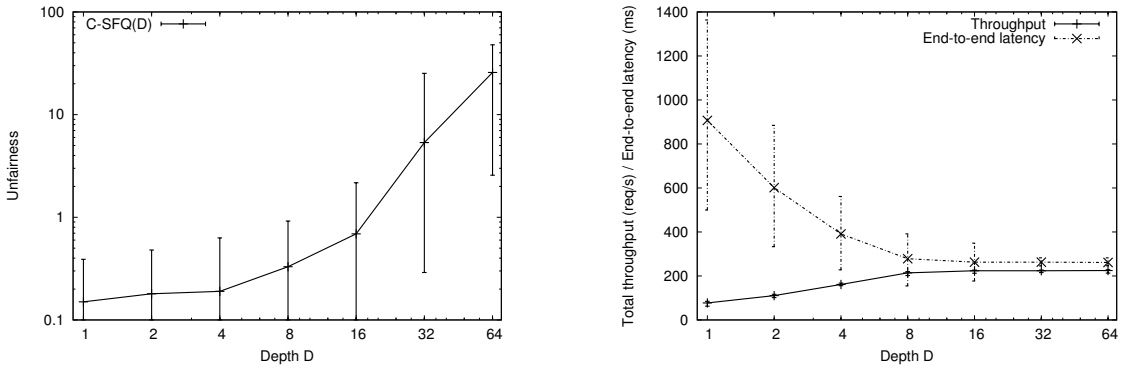

Figure 9: Analysis of the effects of the maximum degree of concurrency D to *C-SFQ(D)*. The graph on the left shows the relation between unfairness and values of D. The graph on the right shows the aggregate throughput and the end-to-end latency obtained from the service for different values of D. In both graphs the x-axis is in logarithmic scale. The dots are the median values and the error bars show the 5th and the 95th percentile of the measurements.

utilization of service resources. It is thus desirable to find a D such that the system is operating at nearly full capacity while unfairness is low at the same time. To find out if there is indeed such a point for our system and workload, the right graph of Figure 9 plots the total throughput of the system as a function of D. We can see that a $D \geq 8$ provides a throughput that is close to the maximum. At this point, the available parallelism inside the 3-tier system is used efficiently. Taking a look at both the graphs in Figure 9, we can see that $D = 8$ or $D = 16$ provide a good operating point, where the throughput is close to its maximum and the unfairness is low.

The trade-off between throughput and unfairness is not the only consideration for the value of D. When the performance goal is latency, then the impact of D on latency has also to be considered. In particular, we consider the *end-to-end latency*, which is the response delay perceived by the clients of the service; this includes any potential waiting time in the scheduler queues. Figure 9 shows the end-to-end latency as a function of D for *C-SFQ(D)*. As there are 75 concurrent clients, the median end-to-end latency goes down as D increases, due to the parallelism inside the 3-tier system. From the graph, we can see that the median of the end-to-end latency reaches its minimum value of approximately 300 ms at $D \geq 8$. This is compatible with the values for D derived from the trade-off between unfairness and throughput. Note that if the service were run without any scheduler in front of it, then the mean latency experienced by the 75 clients would have been approximately 300 ms.

Service time, the time it takes for a request to be processed in the service, is another metric that needs to be considered for this trade-off. According to Little's law, the service time would increase with the value of D. If there is a latency goal, then the value of D is driven by how aggressive that goal is. For example, if the latency goal is 20 ms, then D should be at most 2 in our system. Such a low value for D would result in low unfairness, but would also result in low aggregate throughput according to Figure 9.

5.4 Controlling SFQ(D) and C-SFQ(D)

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 a controller to automatically adjust weights according to system and workload dynamics.

To automatically adjust the scheduler parameters, we designed a feedback loop that reacts to the observed performance of the service (latency and throughput). The feedback loop is not intrusive to the service. It uses an *adaptive dual MIMO controller* [4] which does not require an off-line system identification. Instead, it estimates a model of the black-box service on-line adapting to system non-linearities using recursive least-squares estimation. The model used is of the following form: $y(k) = A_1 y(k-1) + A_2 y(k-2) + B_1 u(k-1) + B_2 u(k-2)$ where $y(k)$ is a column vector of the performance measurements at time k, $u(k-1)$ is a column vector with the ϕ and D settings at time $k-1$, and A_1, A_2, B_1 and B_2 are the estimated model parameter matrices. The control problem is formulated as an LQR problem [14] in which the minimum energy of $|y_{ref}(k) - y(k)|$ is the desired operating point. Here y_{ref} is a column matrix with the desired performances of all the flows. A closed-form control law that minimizes the expected value of this difference, assuming the aforementioned model estimates, can then be found. This law is used to compute the new ϕs and D to set in the scheduler. The closed-loop system is provably stable. The details of the controller and closed-loop design are outside the scope of this paper.

Figure 10 plots the throughput, end-to-end latency, unfairness and actual weight settings over a 90-second time window of the execution. There are two main observations to make. First, the actual performance is close to the goals for both throughput and latency 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 is it completely off the goal. Second, during the periods in which the performance is far from the goal for *SFQ(D)*, the unfairness is high. As shown in the fourth graph, the unfairness peaks at 55, 70 and especially around 100 seconds. This shows that effective differentiation is crucial for successfully controlling the system to achieve performance goals.

The closed-loop design we consider for the experiments here, the controller is executed every 3 seconds. A choice of an interval shorter than that would result in an unstable system, due to measurement noise. This is a typical design constraint for a controlled system. Thus, work conservation is a useful scheduler property, that results in efficient use of service resources in the presence of transient load changes with duration less than an interval.

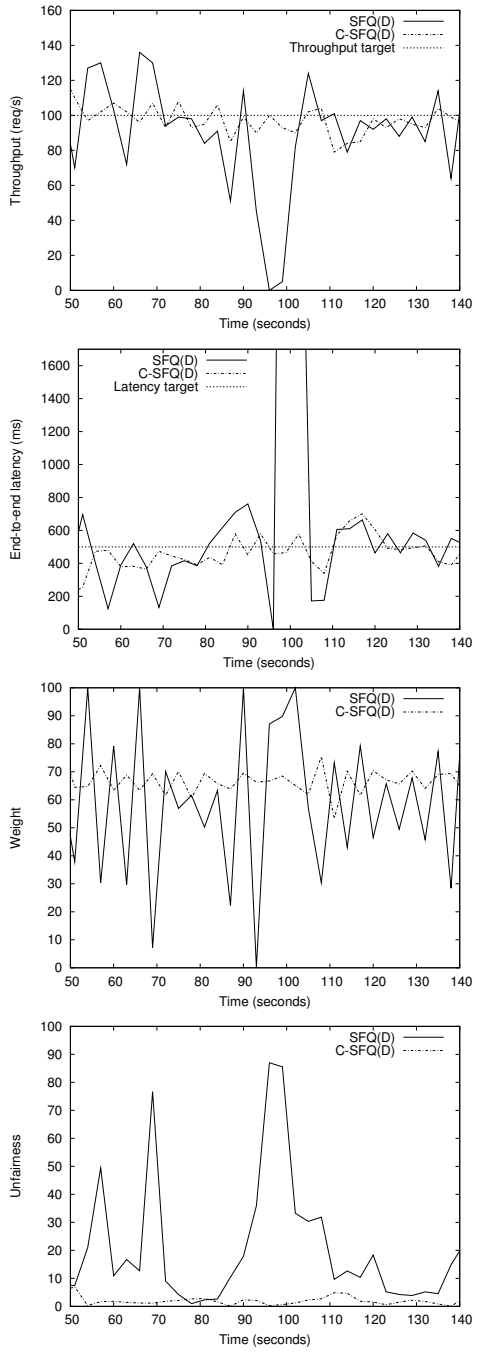

Figure 10: Demonstrating the ability of *SFQ(D)* and *C-SFQ(D)* to meet performance goals. In all these figures, a feedback-based adaptive controller sets the weights every 3 seconds. The four graphs show throughput, latency, weight settings and unfairness respectively. The x-axis refers to the execution time of an actual run.

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 [3, 29]. 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 [3]. 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 [29]. A weighted-fair queuing scheduler is used to enforce the reservations [27]. The learning algorithm is computationally expensive. Thus, reservations can not be adjusted at very fine time granularity. Existing scheduling algorithms have been designed and analyzed for shares that are fixed over time. 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.

Request scheduling externally to the target system has been proposed as an approach to perform resource control for black-box storage systems. Facade [25] proposes a storage switch that uses Earliest Deadline First (EDF) scheduling to meet response latency goals exposed directly to the scheduler. The key drawback of EDF is that it cannot ensure isolation among flows and thus it cannot meet any performance goals in the presence of overload. SLEDS [9] and Triage [23] are two approaches developed for ensuring performance isolation and differentiation among flows. They both use feedback about observed performance to adjust dynamically the flow reservations. Reservations result in low service utilization—spare system capacity cannot be re-used by active flows between reservation adjustments.

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 [13], FQS [18], FFQ [28], WF2Q [7], WF2Q+ [8], SCFQ [12, 15], and SFQ [17]. Fair queuing has been adapted to other contexts such as disk scheduling [27], CPU scheduling [16], and server resource management [30, 24]. Other approaches to implementing proportional sharing include lottery scheduling [31]. Share schedulers are often associated with with resource control abstractions such as Resource Containers [5] or virtual machines [6].

More recently, a family of depth-controlled *WFQ* schedulers have been proposed for proportional sharing of a computing services [21]. They have shown that with appropriate weight settings a *WFQ* scheduler can be used to meet performance goals such as maximum response latencies.

This paper extends this work on proportional-share schedulers by addressing the problem of dynamically varying the flow weights based on on-line feedback from the system, and exploring the interaction of the scheduler properties with feedback controllers.

7. CONCLUSIONS

We are concerned with the problem of enforcing application-level performance goals in a shared computing service, by varying the parameters of an interposed request scheduler. In this paper, we focus on controllability properties of *proportional share schedulers*, which are most commonly implemented using variants of Weighted Fair Queuing (*WFQ*). We prove that existing *WFQ* schedulers are unfair when the flow weights vary. That makes them ineffective in the presence of dynamic control. We define *controllable fairness*, a stronger notion of fairness for this case, we propose a tag adjustment algorithm that ensures that *WFQ* schedulers are controllable-fair, and prove the properties of the resulting schedulers.

To validate the analytical results, we performed an experimental evaluation using a three-tier Web service. We confirm that a typical depth-controlled *WFQ* scheduler, *SFQ(D)*, exhibits poor fairness when flow weights vary by as little as 10%. 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 demonstrate that *C-SFQ(D)* can indeed be used with a feedback controller to enforce performance goals. Finally, we perform a sensitivity analysis of the controllable fairness of *C-SFQ(D)* against the values and deltas of flow weights and degree of concurrency, which shows that *C-SFQ(D)* can be used even with aggressive controllers.

The results of this paper are promising. They indicate that *controllable-fair schedulers* can be used with feedback controllers to meet performance goals for workloads of shared services. The design and analysis of control systems for complex computing services is the topic of future work.

8. REFERENCES

[1] T. Abdelzaher, K. G. Shin, and N. Bhatti. User-level QoS-adaptive resource management in server end-systems. *IEEE Transactions on Computers*, 52(5), 2003.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *International Conference on File and Storage Technologies (FAST)*, pages 175–188, Monterey, CA, January 2002.

[3] M. Aron. *Differentiated and Predictable Quality of Service in Web Server Systems*. PhD thesis, Computer Science Department, Rice University, 2000.

[4] K. J. Åström and B. Wittenmark. *Adaptive Control*. Electrical Engineering: Control Engineering. Addison-Wesley Publishing Company, 2 edition, 1995. ISBN 0-201-55866-1.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[6] P. Barham, B. Dragovic, K. Faser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[7] J. Bennett and H. Zhang. WF2Q: Worst-case Fair Weighted Fair Queueing. In *IEEE Infocom*, pages 120–128, San Francisco, CA, March 1996.

[8] J. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networks*, 5(5):675–689, October 1997.

[9] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtulization for large-scale storage systems. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 109–118, Florence, Italy, October 2003.

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

[11] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to systems states: A building block for automated diagnosis and control. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page xxx, San Francisco, CA, December 2004.

[12] J. Davin and A. Heybey. A simulation study of fair queuing and policy enforcement. *Computer Communications Review*, 20(5):23–29, October 1990.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queuing Algorithm. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, pages 1–12, Austin, TX, September 1989.

[14] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley Publishing Company, 3 edition, 1998. ISBN 0-201-82054-4.

[15] S. J. Golestani. A self-clocked fair queuing scheme for high-speed applications. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, pages 636–646, Toronto, Canada, April 1994.

[16] P. Goyal, X. Guo, and H. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, October 1996.

[17] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networks*, 5(5):690–704, October 1997.

[18] A. Greenberg and N. Madras. How fair is fair queuing? *Journal of the ACM*, 39(3):568–598, July 1992.

[19] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, pages 14–24, New York, NY, June 2004.

[20] *Java PetStore*. www.middleware-company.com.

[21] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, pages 37–48, New York, NY, USA, June 2004.

[22] A. Kamra, V. Misra, and E. Nahum. Yaksha: A Self-Tuning Controller for Managing the Performance of 3-Tiered Web sites. In *International Workshop on Quality of Service (IWQoS)*, pages 47–56, Montreal, Canada, June 2004.

[23] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. In *International Workshop on Quality of Service (IWQoS)*, pages 67–74, Montreal, Canada, June 2004.

[24] Z. Liu, M. Squillante, and J. Wolf. On maximizing service-level-agreement profits. In *ACM Conference on Electronic Commerce (EC)*, pages 213–223, New York, NY, October 2001.

[25] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. In *International Conference on File and Storage Technologies (FAST)*, pages 131–144, San Francisco, CA, March 2003.

[26] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 225–238, Boston, MA, December 2002.

[27] P. J. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, pages 44–55, Madison, WI, June 1998.

[28] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networks*, 6(5):611–624, October 1998.

[29] V. Sundaram and P. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *International Workshop on Quality of Service (IWQoS)*, pages 479–497, Monterey, CA, June 2003.

[30] B. Urgaonkar, P. J. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 239–254, Boston, MA, December 2002.

[31] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, November 1994.

[32] Z.-L. Zhang, D. F. Towsley, and J. F. Kurose. Statistical analysis of generalized processor sharing scheduling discipline. *IEEE Journal on Selected Areas in Communications*, 13(6):1071–1080, 1995.