# Implementation of SRPT Scheduling in Web Servers

Mor Harchol-Balter Nikhil Bansal Bianca Schroeder

Nov 2000
CMU-CS-00-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

This paper proposes a method for improving the performance of Web servers servicing static HTTP requests. The idea is to give preference to those requests which are quick, or have small remaining processing requirements, in accordance with the SRPT (Shortest-Remaining-Processing-Time) scheduling policy.

The implementation is at the kernel level and involves controlling the order in which socket buffers are drained into the network. Experiments use the Linux operating system and the Flash web server. All experiments are repeated under a range of server loads and under both trace-based workloads and those generated by a Web workload generator.

Results indicate that SRPT-based scheduling of connections yields significant reductions in mean response time, mean slowdown, and variance in response time at the Web server. Most significantly, and counter to intuition, the *large requests* are only negligibly penalized (or not at all penalized) as a result of SRPT-based scheduling.

**Keywords:** Web servers, connection-scheduling, HTTP requests, priority, M/G/1, shortest processing remaining time, starvation, heavy-tailed workloads, time-sharing.

# 1 Introduction

Today's busy Web servers may be servicing hundreds of requests at the same time. This can cause large queueing delays at the Web server, or close to it. Our overall goal in this paper is to minimize the queueing delay at a Web server.

The idea is simple. Recent measurements [21] have suggested that the request stream at most Web servers is dominated by *static* requests, of the form "Get me a file." For such requests, the *size of the request* (i.e. the time required to service the request) is well-approximated by the size of the file, which is well-known to the server. Thus far, (almost) no companies or researchers have made use of this information. We propose to use the knowledge of the size of the request to affect the scheduling order in which requests are serviced by the Web server, and in this way minimize the queueing delay at the Web server.

Traditionally, requests at a Web server are scheduled independently of their size. The requests are time-shared, with each request receiving a *fair share* of the Web server resources. We propose, instead, *unfair scheduling*, in which priority is given to *short* requests, or those requests which have *short remaining time*, in accordance with the well-known scheduling algorithm Shortest-Remaining-Processing-Time-first (SRPT).

There is an obvious reason why *unfair scheduling* is not used. Unfair scheduling seems to imply that some requests will "starve," or at least be harshly penalized (see Section 2 for a list of references to this effect). This intuition is usually true. However, we have a new theoretical paper, [22], which proves that in the case of (heavy-tailed) Web workloads, this intuition falls apart. In particular, for Web workloads, even the largest requests are *not* penalized (or negligible penalized) by SRPT scheduling. These new theoretical results have motivated us to reconsider "unfair" scheduling.

It's not immediately clear what SRPT means in the context of a Web server. The SRPT scheduling policy is well-understood in the context of a single-queue, single-resource system [27]: at any moment in time, give the full resource to that one request with the shortest remaining processing time requirement. However a Web server is not a single-resource system; thus it would be highly inefficient to schedule only *one* request at a time to run in the Web server. Furthermore, it is not even obvious *which* of the Web server's resources need to be scheduled.

After experimenting with various Web servers and various Web workload generators, we've reached the conclusion that for Web servers the network is the *bottleneck resource*. Access links to Web sites (T3, OC3, etc.) cost thousands of dollars per month, whereas CPU is cheap in comparison. Thus the CPU is never the bottleneck (since it is inexpensive to add more CPU). Likewise disk utilization remains low since most files end up in the cache. In this paper we use a 10Mb/sec link out of our Web server, because it is easy to saturate the link in experimentation. While the link bandwidth is saturated, the CPU utilization and disk utilization remain extremely low.

Since the network is the bottleneck resource, we try to apply the SRPT idea at the level of the network. Our idea is to control the order in which socket buffers are drained. Recall that for each (non-persistent) request a connection is established between the client and the Web server, and corresponding to each connection, there is a socket buffer on the Web server end into which the Web server writes the contents of the file requested. Traditionally, the different socket buffers are drained in Round-Robin Order (each getting a fair share of the bandwidth of the outgoing link). We instead propose to give priority to those sockets corresponding to connections for small file requests or where the *remaining data* required by the request is small. In Sections 3 and Sections 8, we describe the high-level and low-level issues, respectively, involved in implementing priority scheduling of socket buffers.

Our experiments use the Linux operating system and the Flash Web server [23]. We chose the Linux o.s. because of its popularity. We could have used any Web server, since our modifications are primarily in the o.s., but we chose Flash simply because it was easy to work with (substantially fewer lines of code compared with other Web servers). In order to perform statistically meaningful experiments, our clients use a request sequence taken from a Web trace, or generated by a Web workload generator (See Section 4.2). This request sequence is controlled so that the server load remains below 1 and so that the same experiment can be repeated at different server loads (the server load is the load at the bottleneck device – in this case the network link out of the Web server). The experimental setup is detailed in Section 4.

Each experiment is repeated in two ways:

- Under the standard Linux o.s. (fair-share draining of socket buffers) with an unmodified Web server. We call this **fair scheduling**.
- Under the modified Linux o.s. (SRPT-based draining of socket buffers) with the Web server modified only to update socket priorities. We call this **SRPT-based scheduling**.

We experiment with different Web workloads, different system loads, and different lengths of experiments. For each experiment we measure mean response time, variance in response time, mean slowdown, and variance in slowdown. We also measure the mean response time as a function of the request size, to examine the question of whether the mean response time for just the largest requests is higher under SRPT-based scheduling as compared with fair scheduling. We find the following results:

- SRPT-based scheduling decreases mean response time by a factor of 2 – 5 for loads greater than $0.5$.
- SRPT-based scheduling decreases the mean slowdown by a factor of 2 – 7 for loads greater than $0.5$.
- SRPT-based scheduling helps small requests a lot, while negligibly penalizing large requests. For example, under a load of $0.8$, our experiments show that $80\%$ of the requests improve by a factor of close to 10 under SRPT-based scheduling, with respect to mean response time, and all but the top $0.5\%$ of the requests improve by a factor of over 5 under SRPT-based scheduling, with respect to mean response time. In contrast, the very largest request suffers an increase in mean response time under SRPT-based scheduling of a factor of only 1.2.
- The variance in the mean response time for most requests under SRPT-based scheduling is far lower for *all* requests, in fact several orders of magnitude lower for most requests.
- *SRPT-based scheduling (as compared with FAIR scheduling) does not have any effect on the network throughput or the CPU utilization.*

The results are detailed in Section 5 and are in agreement with theoretical predictions (see Section 6.1).

In Section 3 we take a closer look at the Linux kernel internals in an attempt to explain why the fair scheduling experiment resulted in such poor performance. We find that although the Linux kernel is supposed to implement fair scheduling, the particular structure of queues in the Linux kernel inadvertently creates some bias against *small requests*, thus being *unfair* to *small requests*.

The above observations about Linux lead us to propose a "quick-fix" to Linux, with the following results:

- The performance of the smallest $50\%$ of the requests improves by a factor of 10 with respect to mean response time under the "quick-fix".
- The largest $50\%$ of the requests are not negatively impacted whatsoever by the "quick-fix." In fact, many benefit.

The "quick-fix" and its results are discussed in Section 7.

It is important to realize that this paper is just a prototype to illustrate the power of using SRPT-based scheduling. In Section 10, we elaborate on broader applications of SRPT-based scheduling, including the application of SRPT-based scheduling to other resources, e.g. CPU, to cgi-scripts and other non-static requests, and finally the application of SRPT-based scheduling to routers throughout the Internet.

## 2 Relevant Previous Work

We first discuss related implementation work and then discuss relevant theoretical results.

Many recent papers have dealt with the issue of how to obtain differentiated quality of service in Web servers, via priority-based scheduling of requests. These papers are generally interested in providing different levels of QOS to different customers, rather than using a size-based scheme like our own. Various ideas have been tried to implement such prioritization schemes. We describe these below.

Almeida et. al. [2] use both a user-level approach and a kernel-level implementation to prioritizing HTTP requests at a Web server. The *user-level* approach in [2] involves modifying the Apache Web server to include a Scheduler process which determines the order in which requests are fed to the Web server. This modification is all in the application level and therefore does not have any control over what the o.s. does when servicing the requests. The *kernel-level* approach in [2] simply involves setting the priority of the process which handles a request in accordance with the priority of the request. Observe that setting the priority of a process only allows very coarse-grained control over the scheduling of the process, as pointed out in the paper. The user-level and kernel-level approaches in this paper are good starting points, but the results show that more fine-grained implementation work is needed. Specifically, the high-priority requests only benefit by up to $20\%$ and the low priority requests suffer by up to $200\%$.

Another attempt at priority scheduling of HTTP requests is more closely related to our own because it too deals with SRPT scheduling at Web servers [11]. This implementation does not involve any modification of the kernel. The authors experiment with connection scheduling at the *application level* only. They design a specialized Web server which allows them to control the order in which `read()` and `write()` calls are made, but does not allow any control over the low-level scheduling which occurs inside the kernel, below the application layer (*e.g.*, control over the order in which socket buffers are drained). Via the experimental Web server, the authors are able to improve mean response time by a factor of close to 4, for some ranges of load, but the improvement comes at a price: a drop in throughput by a factor of almost 2. The explanation, which the authors offer repeatedly, is that scheduling at the application level does not provide fine enough control over the order in which packets enter the network. In order to obtain enough control over scheduling, the authors are forced to limit the throughput of requests. This will not be a problem in our paper. Since the scheduling is done at the kernel, we have absolute control over packets entering the network. Our performance improvements are greater than those in [11] and do not come at the cost of any decrease in throughput.

The papers above offer coarser-grained implementations for priority scheduling of connections. Very recently, many operating system enhancements have appeared which allow for finer-grained implementations of priority scheduling [15, 25, 3, 4]. In this paper we work with one of these implementations, known as the Diffserv patch, as described in Section 3.

Several papers have considered the idea of SRPT scheduling in theory.

Bender, Chakrabarti, and Muthukrishnan [8] consider size-based scheduling in Web servers. The authors reject the idea of using SRPT scheduling because they prove that SRPT will cause large files to have an arbitrarily high *max slowdown*. However, that paper assumes a worst-case adversarial arrival sequence of Web requests. The paper goes on to propose other algorithms, including a theoretical algorithm called Dynamic Earliest Deadline First (DEDF), which does well with respect to max slowdown and mean slowdown.

Roberts and Massoulie [26] consider bandwidth sharing on a link and survey various scheduling policies. They suggest that SRPT scheduling may be beneficial in the case of a heavy-tailed (Pareto) flow sizes.

The primary theoretical motivation for this paper, comes from our own paper, [22]. This is a theoretical paper on the starvation properties of SRPT scheduling. The authors prove bounds on how much worse a request could perform under SRPT scheduling as compared with PS (processor-sharing, a.k.a. fair-share time-sharing) scheduling, within an M/G/1 setting. The authors also corroborate their results using a trace-driven simulation with real arrival stream. The authors prove that the penalty to large requests under SRPT (as compared with PS) is not severe. In particular, they show that for a large range of *heavy-tailed* (Pareto) distributions, *every single request*, including the very largest request, performs better under SRPT scheduling as compared with PS scheduling. The case of heavy-tailed request size distributions is important because heavy-tailed distributions have been shown to arise in many empirical computer workloads [20, 17, 9, 19, 24]. In particular measurements of *Web file sizes* and *HTTP request times* have been shown to be heavy-tailed [7]. We use the theoretical results in [22] to corroborate the results in this paper.

The general idea of size-based scheduling for heavy-tailed workloads has also been explored in arenas other than Web servers. Shaikh, Rexford, and Shin [28] discuss routing of IP flows (which have heavy-tailed size distributions) and propose routing long flows differently from short flows. Harchol-Balter [18] considers scheduling in distributed server systems where requests are not preemptible, request sizes are unknown, and where the workload is heavy-tailed. She proposes size-based scheduling using an algorithm for gradually learning sizes.

# 3 Implementation of prioritized socket draining: high-level description

In Section 3.1 we explain how socket draining works in standard Linux. In Section 3.2 we describe an existing patch for Linux versions 2.2 and above, known as the Diffserv patch [1]. This patch is supposed to enable the draining of sockets in a prioritized fashion. Socket priorities are actually derived, set, and dynamically updated by the Web server. Section 3.3 describes the implementation end at the Web server and also deals with the algorithmic issues of determining which file sizes should fall within which priority classes.

The implementation of prioritized socket draining in Linux turned out to be a more involved process than we initially believed. For readability, we postpone a discussion of these low-level issues to Section 8. In Section 8.1 we describe our many failed attempts at getting the Diffserv patch to work, and the fixes that we finally came up with. Obtaining a priority capability alone, however, is not enough to make size-based queueing work. One problem is that for small requests, a large portion of the time to service the request is spent *before* the size of the request is even known. Section 8.2 describes our solution to this problem.

## 3.1 Default Linux configuration

Figure 1 shows our understanding of the flow of control in standard Linux. This understanding was obtained via extensive experiments which we don't have room to describe here. Observe that the model that we describe in this section and in the subsequent section is *not* visible by looking at the Linux code alone.

There is a socket buffer corresponding to each connection. Data streaming into each socket buffer is encapsulated into packets which obtain TCP headers and IP headers. Throughout this processing, the packet streams corresponding to each connection is kept separate. Finally, there is a *single* "priority queue", into which *all* streams drain *fairly*. Our experiments show that this single "priority queue," although bounded, can get quite long. Packets leaving this queue drain into a short Ethernet card queue and out to the network.

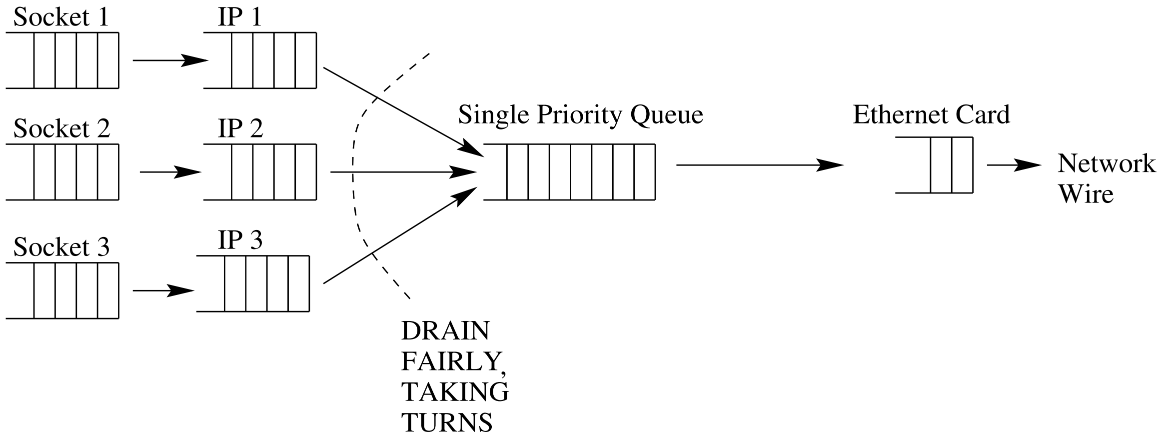



Figure 1: *Flow of control in Standard Linux. The important thing to observe is that there is a* single *priority queue into which all connections drain fairly.*

## 3.2 How Linux with Diffserv patch works

Figure 2 shows our understanding of the flow of control in Linux under Diffserv patch[1]. This understanding was obtained via extensive experiments and by reading the following papers: [15, 25, 3, 4].

[1]We used the Diffserv patch [1] in our experiments. More recent versions of Linux include priority scheduling as an option, although this mechanism needs to be enabled via `tc` in order for the priorities to work.

Again, there is a socket buffer corresponding to each connection. Data streaming into each socket buffer is encapsulated into packets which obtain TCP headers and IP headers. Throughout this processing, the packet streams corresponding to each connection is kept separate. Finally, there are 16 priority queues. These are called bands and they range in number from 0 to 15, where band 15 has lowest priority and band 0 has highest priority. All the connections of priority $i$ drain fairly into the $i$th priority queue. The priority queues then drain in a prioritized fashion into the Ethernet Card queue. Priority queue $i$ is only allowed to drain if priority queues 1 through $i-1$ are all empty. Again we note that our experiments indicate that the priority queues can become quite long in practice.

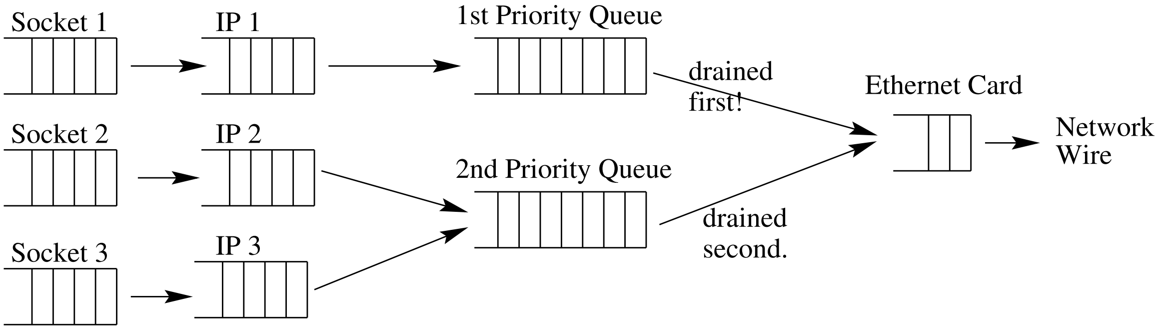



Figure 2: *Flow of control in Linux with Diffserv Patch. It is important to observe that there are several priority queues, and queue $i$ drains only if all of queues $0$ through $i-1$ are empty.*

### 3.3 Setting of priorities in the Web server and Algorithmic issues in approximating SRPT

Very few changes must be made to the Web server itself. The socket priorities must be initialized (using the `setsockopt()` system call) based on the initial size of the request. Later the socket priorities must be updated, in agreement with the remaining size of the file.

SRPT assumes infinite precision in ranking the remaining processing requirements of requests. In practice, we are limited to a small fixed number of priority bands (16).

It turns out that the way in which request sizes are partitioned among these priority levels is important with respect to the performance of the Web server. In the experiments throughout this paper, we use only 5 priority classes to approximate SRPT. Our choice of size cutoffs in this paper is rather ad hoc. We believe that more optimal size cutoffs exist, but we didn't have time to do a proper search.

Our SRPT-like algorithm is thus as follows:

1. Priorities 1, 2, 3, 4, and 5 are associated with size ranges, where 1 denotes highest priority.

2. When a request arrives, it is given a socket with priority 0. This is an *important* detail which allows SYN ACKS to travel quickly. This is explained in depth in Section 8.2.

3. After the request size is determined (by looking at the URL of the file requested), the priority of the socket corresponding to the request is reset based on the size of the request.

4. As the remaining size of the request diminishes, the priority of the socket is dynamically updated to reflect the remaining size of the request.

# 4 Experimental Setup

## 4.1 Architecture

Our experimental architecture involves two machines each with an Intel Pentium III 700 MHz processor and 256 MB RAM, running Linux 2.2.14, and connected by a 10Mb/sec full-duplex Ethernet connection. The Flash [23] Web server is running on one of the machines. The other machine hosts up to 200 clients which send requests to the Web server.

## 4.2 Workload

The clients' requests are generated either via a *Web workload generator* (we use a modification of `Surge` [7]) or via *traces*. *Throughout this paper, all results shown are for a trace-based workload.* We have included in the Appendix the *same*, *full set* of results for the `Surge` workload. We have chosen to deemphasize the `Surge`-based results because we discovered some shortcomings in the `Surge` workload that made us question the `Surge`-based results.

### 4.2.1 Traces

The traces come from the Soccer World Cup 1998, and were downloaded from the Internet Traffic Archive [16]. We used only 7 minutes of the trace (from 10:00 p.m. to 10:07 p.m.). We incorporated only static requests in our experiment.

Some statistics about our trace workload: The mean file size requested is 5K bytes. The min size file requested is a 41 byte file. The max size file requested is a 2.020644 MB file. There are approximately 90,000 requests made, which include requests for over a thousand *different* files. The distribution of the file sizes requested fits a heavy-tailed Pareto distribution. The the largest $< 3\%$ of the requests make up $> 50\%$ of the total load, exhibiting a strong heavy-tailed property. $50\%$ of files have size less than 1K bytes. $90\%$ of files have size less than 9.3K bytes.

### 4.2.2 Web workload generator

We also repeated all experiments using a Web workload generator. These results are shown in the Appendix.

### 4.2.3 Determination of System Load

As mentioned earlier, it is important that we understand the system load under which each experiment is being run. We have derived a formula for system load. For additional verification, this formula was derived in two independent ways.

Since the bandwidth is the bottleneck resource, we define the system load to be the ratio of the bandwidth used on average and the maximum bandwidth available . Thus if our arrival sequence is such that 8Mb of bandwidth is utilized on a 10Mb/s link, we say that our system is running at a load of 0.8.

We start by describing the analytic method for finding the arrival rate which corresponds to a desired load. This is very simple. Since we know the average number of bytes per request, we can estimate the fraction of the total bandwidth that would be used under any arrival rate.

However this method does not allow us to obtain a very accurate estimate of load, since the data transfered over a link includes the size of the various network protocol headers. Moreover we cannot be sure that the maximum bandwidth available is exactly 10Mb/s. So our load estimate might be inaccurate.

To obtain a very accurate estimate of the load, we start with a small arrival rate and measure the network bandwidth utilization (rather than calculating it). Then we increase the rate slowly and again measure the bandwidth. The bandwidth increases linearly with the arrival rate, but then stops increasing when the arrival rate reaches a value at which the system load first becomes $1$. We note this critical arrival rate. To obtain a load $\rho < 1$ we just set the arrival rate to $\rho$ times the critical arrival rate.

Throughout our experiment, we verify that our estimates for loads using all the methods agree, since even a small difference (say 0.8 as opposed to 0.9) can create a lot of difference in the results.

# 5 Experiments and Experimental Results

We run a series of experiments comparing:

**Standard Linux with FAIR Scheduling** versus **Linux with our SRPT-based Priority Scheme**

Each experiment is run for 10 minutes. For each experiment, we evaluate the following performance metrics:

- *Mean response time*. The response time of a request is the time from when the client submits the request until the client receives the last byte of the request.
- *Mean slowdown*. The slowdown metric attempts to capture the idea that clients are willing to tolerate long response times for large file requests and yet expect short response times for short requests. The slowdown of a request is therefore its response time divided by its size in bytes. Slowdown is also commonly known as *normalized response time* and has been widely used [13, 5, 17]. Mean slowdown is the average of the slowdown of each of the requests.
- *Mean response time as a function of request size*. This will indicate whether big requests are being treated *unfairly* under SRPT as compared with FAIR-share scheduling.
- *Coefficient of variation of response time as a function of request size*.

Before presenting the results of our experiments, we make some important comments.

- In all of our experiments the network was the bottleneck resource. CPU utilization during our experiments ranged from $1\%$ in the case of low load to $5\%$ in the case of high load.
- The measured throughput and bandwidth utilization under the experiments with SRPT scheduling is *identical* to that under the same experiments with FAIR scheduling, and in both cases the maximum possible throughput is achieved, given the request sequence.
- The same exact set of requests complete under SRPT scheduling and under FAIR scheduling.

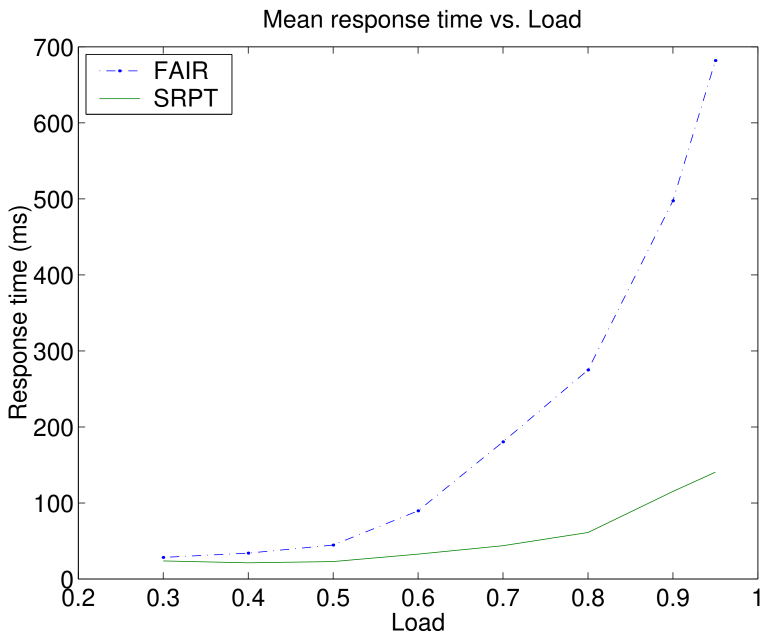



Figure 3: *Mean response time under SRPT scheduling versus traditional FAIR scheduling as a function of system load, under trace-based workload.*

- There is no additional CPU overhead involved in SRPT scheduling as compared with FAIR scheduling. This can be explained by two observations. First, the overhead due to updating priorities of sockets is insignificant, given the small number of priority classes that we use. Second, the preemption overhead under SRPT-based scheduling is actually *lower* than under FAIR scheduling. See [22] for a proof of this fact.

Figure 3 shows the mean response time under SRPT scheduling as compared with the traditional FAIR scheduling as a function of load. For lower loads the mean response times are the same under the two scheduling policies. However for loads $> 0.5$, the mean response time is a factor of 2 – 5 lower under SRPT scheduling. These results are in agreement with our theoretical predictions in [22].

The results are even more dramatic for mean slowdown. Figure 4 shows the mean slowdown under SRPT scheduling as compared with the traditional FAIR scheduling as a function of load. For lower loads the slowdowns are the same under the two scheduling policies. However for loads $> 0.5$, the mean slowdown is a factor of 2 – 7 lower under SRPT-based scheduling as compared with FAIR scheduling.

The important question is whether the significant improvements in mean response time come at the price of significant unfairness to large requests. Figure 5 shows the mean response time as a function of request size, in the case where the load is $0.6$, $0.8$, and $0.9$. In the left column of Figure 5, request sizes have been grouped into 60 bins, and the mean response time for each bin is shown in the graph. The 60 bins are determined so that each bin spans an interval $[x, 1.2x]$. It is important to note that the last bin actually contains only one request. Observe that small requests perform far better under SRPT scheduling as compared with FAIR scheduling, while large requests, those $> 1$ MB, perform only negligibly worse under SRPT as compared with FAIR scheduling. For

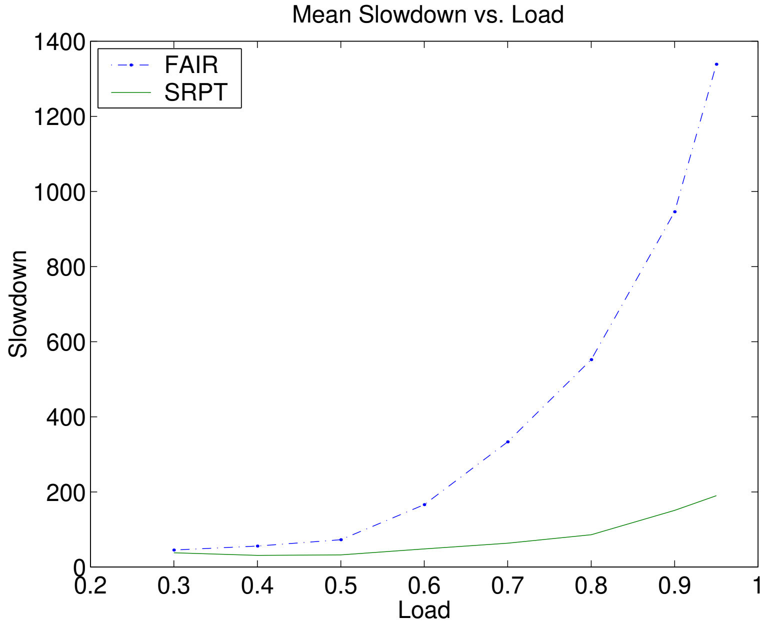
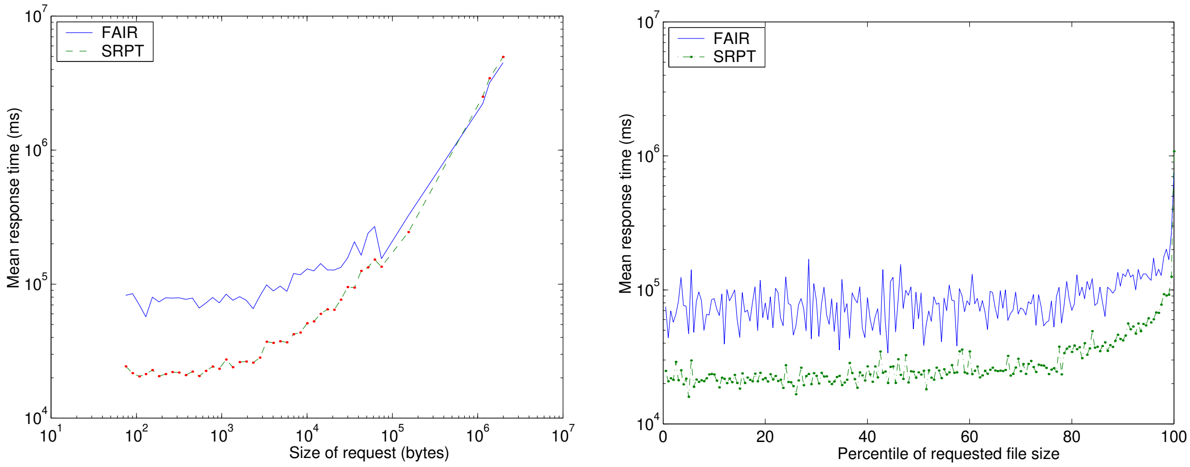



Figure 4: *Mean slowdown under SRPT scheduling versus traditional FAIR scheduling as a function of system load, under trace-based workload.*

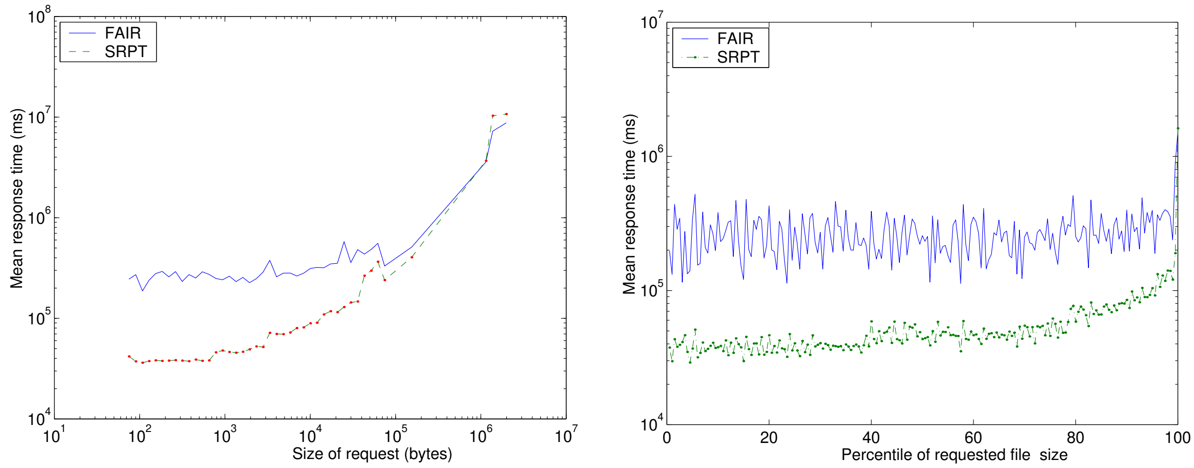
example, under load of $0.8$ (see Figure 5(b)) SRPT scheduling improves the mean response times of small requests by a factor of close to $10$, while the largest request is penalized by a factor of only $1.2$. The right column of Figure 5 is identical in content to the left column, but this time we see the mean response time as a function of the percentile of the request size distribution, in increments of half of one percent (i.e. 200 percentile buckets). From this graph, it is clear that in fact at least $99.5\%$ of the requests benefit under SRPT scheduling. In fact, the $80\%$ smallest requests benefit by a factor of $10$, and all requests outside of the top $1\%$ benefit by a factor of $> 5$. For lower loads, the difference in mean response time between SRPT and FAIR scheduling decreases, and the unfairness to big requests becomes practically nonexistent. For higher loads, the difference in mean response time between SRPT and FAIR scheduling becomes greater, and the unfairness to big requests also increases. Even here though, it is only the top half of one percent of all requests which have worse performance under SRPT, as compared with FAIR scheduling.

The most dramatic improvements of SRPT are in the area of variance reduction. Figure 6 shows the variance in response time for each request size and the coefficient of variation in response time for each request size, as a function of the percentile of the request size distribution. This figure shows the case of load equal to $0.8$. The improvement under SRPT with respect to variance in response time is 3 orders of magnitude for the $80\%$ smallest files; 2 orders of magnitude for files in the $80$th to $99.5$th percentiles and 1 order of magnitude for files in the top $0.5\%$-tile. The results for coefficient of variation are similar, except that now the improvement is by 1.5 orders of magnitude, rather than three.

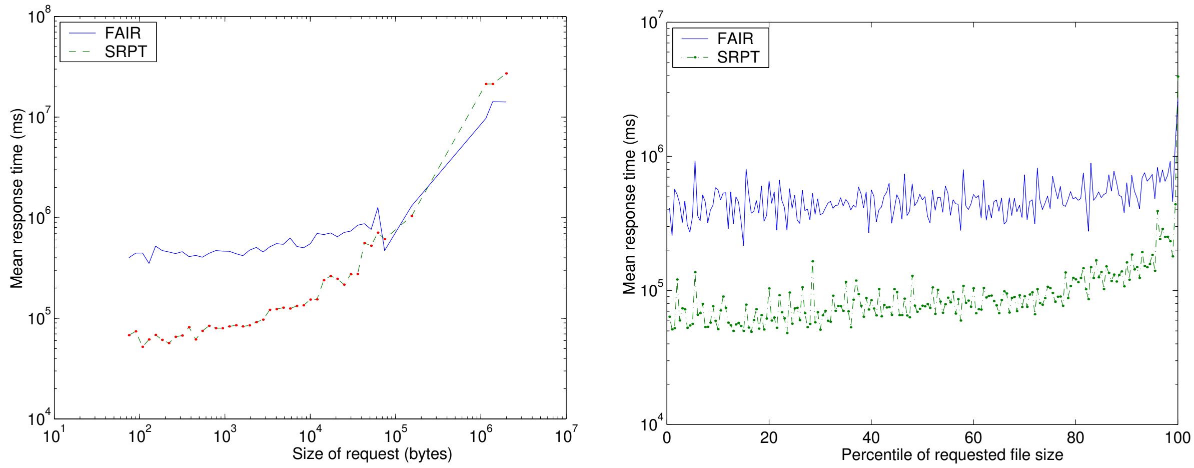
It is interesting to observe that the SRPT results curve is relatively smooth, except for a few odd spikes. The spikes represent occasional packet loss. When packets are lost in a small file, there is a 3 second delay (because RTT has not yet been adjusted). This can throw off the variance quite a bit.




(a) **load = .6**

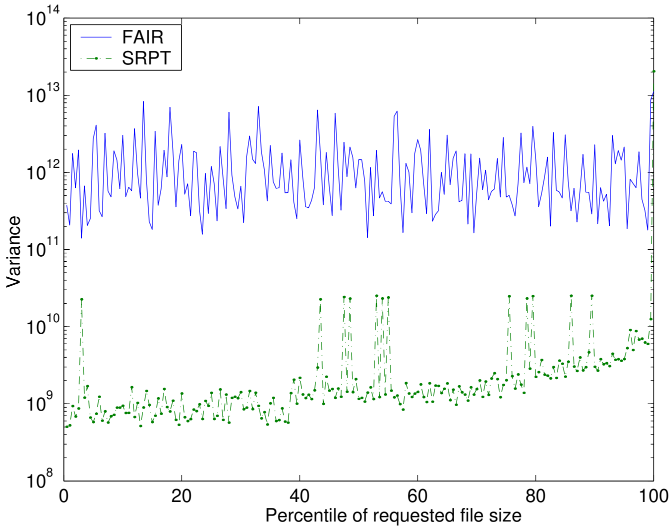



(b) **load = .8**

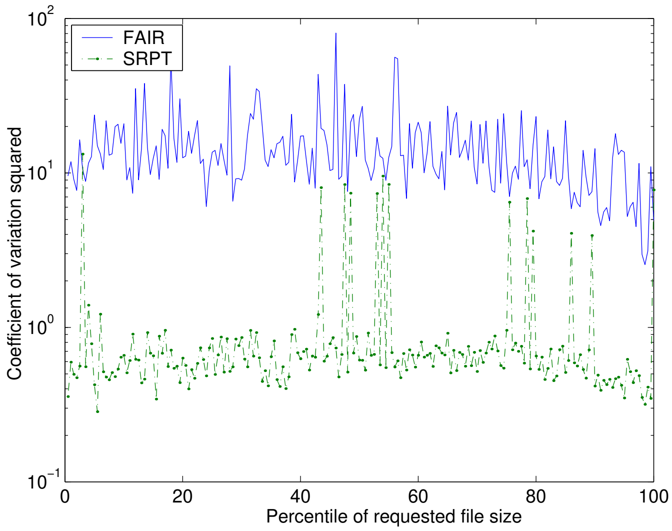



(c) **load = .9**

Figure 5: *Mean response time as a function of request size under trace-based workload, shown for a range of system loads. The left column shows the mean response time as a function of request size. The right column shows the mean response time as a function of the percentile of the request size distribution.*




$(\mathrm{a})$**Variance in response time as a function of request size.**




$(\mathrm{b})$**Squared coefficient of variation of response time as a function of request size.**

Figure 6: *Variance in response time and coefficient of variation of response time as a function of the percentile of the request size distribution for SRPT as compared with FAIR scheduling, under trace-based workload with load* = $0.8$.

# 6 Explanation of Results

The results in the previous section may appear surprising. In this section we offer both a *theoretical* and an *implementation-level* explanation for the previous results.

## 6.1 Theoretical Explanation of Results

It is well-known that the SRPT scheduling policy always produces the minimum mean response time, for any sequence of requests. However, it has also been suspected by many that SRPT is a very unfair scheduling policy for large requests. The above results have shown that this suspicion is false for Web workloads.

It is easy to see why SRPT should provide huge performance benefits for the small requests, which get priority over all other requests. In this section we describe briefly why the large requests also benefit under SRPT, *in the case of a heavy-tailed workload*.

In general a heavy-tailed distribution is one for which

$$\Pr\{X > x\} \sim x^{-\alpha},$$

where $0 < \alpha < 2$. A set of request sizes following a heavy-tailed distribution has some distinctive properties:

1. Infinite variance (and if $\alpha \leq 1$, infinite mean). In practice there is a finite maximum request size, which means that the moments are all finite, but still quite high.
2. The property that a tiny fraction (usually $< 1\%$) of the very longest requests comprise over half of the total load. We refer to this important property as the *heavy-tailed property*.

Request sizes are well-known to follow a heavy-tailed distribution [10, 12]. Thus Web workload generators like `Surge` specifically use a heavy-tailed distribution in their model. Our traces also have strong heavy-tailed properties. (In our trace the largest $< 3\%$ of the requests make up $> 50\%$ of the total load.)

The important property of heavy-tailed distribution is the heavy-tailed property. Consider a large request, in the $99\%$-tile of the request size distribution. This request will actually do much better under SRPT scheduling than under FAIR scheduling for a heavy-tailed workload. The reason is that this big request only competes against $50\%$ of the load under SRPT (the remaining $50\%$ of the load is made up of requests in the top $1\%$-tile of the request size distribution) whereas it competes against $100\%$ of the load under FAIR scheduling. The same argument could be made for a request in the $99.5\%$-tile of the request size distribution. However, it is not obvious what happens to a request in the $100\%$-tile of the request size distribution (i.e. the largest possible request). To understand this, we refer the reader to [22].

## 6.2 Implementation-level Explanation of Results

Section 6.1 concentrated primarily on why the SRPT-based policy performed so well. Another perspective is to ask why the FAIR policy performed so poorly. To see this, consider more carefully Figure 1 which shows flow of control in standard Linux. Observe that all socket buffers drain into the same single priority queue. This queue may grow long (though it is still bounded in length by a Linux parameter). Now consider the effect on a new short request. Since every request has to wait in the priority queue, which may be long, the short request typically incurs a cost of close to 200 ms just for waiting in this queue (assuming high load). This is a very high startup penalty, considering that the service time for a short request should really only be about 20 ms. Our SRPT-based implementation allows short requests to wait in their own separate priority queue which has a very low load and therefore is much shorter. This explains why the response time for short requests improves by close to an order of magnitude under SRPT, as compared with FAIR scheduling[2].

# 7 How to get SRPT-like improvements without using SRPT

The results of the previous section were good, but required a full implementation of the SRPT algorithm. In this section we explore a "quick fix" to Linux. We use only 2 priority bands. All SYN ACKS and all small requests go to the high priority band. All other requests go to the low priority band. We define the cutoff between "small" and "large" such that $50\%$ of the requests are small and $50\%$ are large (note, this is not the same thing as equalizing load, but works better for this case).

We find that this "quick fix" alone is quite powerful. Figure 7 shows the mean response time as a function of file size in the case of system load $0.8$. Compare this figure with the results using the full SRPT implementation, shown in Figure 5(b). The "quick fix" benefits the smallest $50\%$ of requests by a factor of 5, while not harming the large requests at all. This results in a factor of $2.5$ improvement in mean response time and a factor of 5 improvement in mean slowdown. Note that the quick fix only helps $50\%$ of the requests as compared to $99.5\%$ which were helped in the SRPT implementation. Nonetheless, the quick fix still presents significant improvement over traditional FAIR scheduling.

# 8 Implementation of prioritized socket draining: low-level details

## 8.1 Why the Diffserv patch may not work for you, and how to fix it

This section describes some of our attempts at getting the Linux Diffserv patch to work. We hope that this section will be useful to others. All of our experiments involved a 10 Mb/sec Ethernet connection.

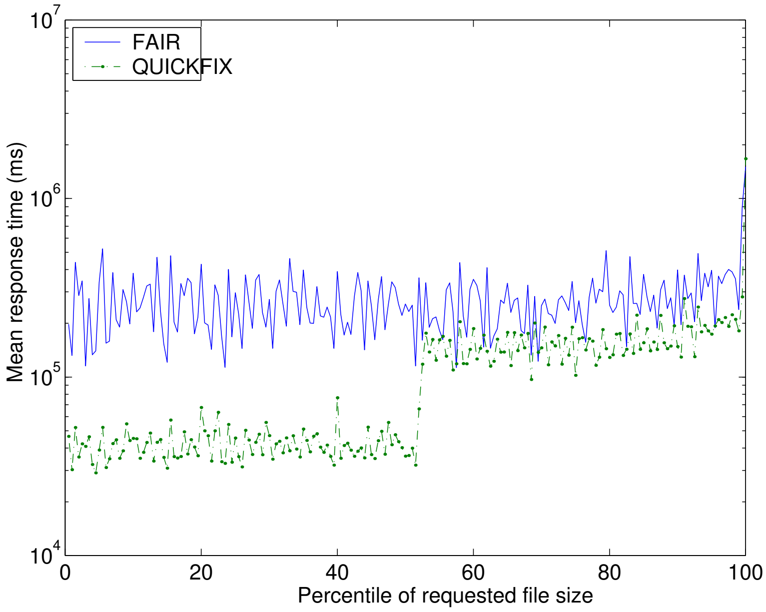
After installing the Diffserv patch, we tried a simple experiment: We opened two TCP connections and flooded both connections with data. Specifically, we repeatedly wrote 1K data into each socket within an infinite write loop (note that the writes are non-blocking). We gave high priority to connection 1 and low priority to connection 2. We expected that the packets on the wire would all be connection 1 packets. In truth, however, only $51\%$ of the packets on the wire were connection 1 packets and $49\%$ were connection 2 packets.

We repeated the above experiment, but this time with two UDP connections and saw a $60\%/40\%$ split between connection 1 packets and connection 2 packets.

We next observed that when we increased the number of connections from 2 to 10, we always achieved the desired $100\%/0\%$ ratio, however we desired a solution that did not require more than 2 connections.

After various other such experiments, we reached the conclusion that the critical parameter in achieving differentiated services is the size of the server's send socket buffer and the client's receive

[2] Observe that the lovely mechanism of [14] which maintains a separate queue for each connection all the way down to the datalink level will likely fix this problem in Linux, if applied at the server end.



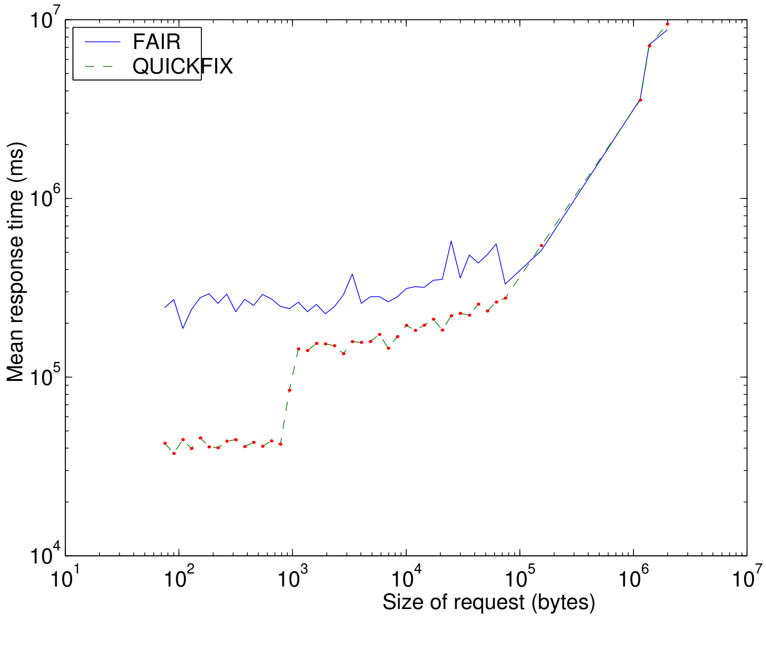




Figure 7: *Mean response time as a function of file size under quick-fix scheduling versus traditional FAIR scheduling, under trace-based workload, with load* $0.8$*.*

socket buffer. All the experiments in this paper using Linux Diffserv have been run with the send socket buffer increased by a factor of 3 and the receive socket buffer increased by a factor of 1.5 (from 64K to 100K). In this mode, we are able to get *full* priority scheduling ($100\%/0\%$ ratio of connection 1 packets to connection 2 packets).

### 8.2 Further changes necessary to make size-based algorithms show effect: the problem of high startup overhead

Having made the adjustments described in the previous section, it seems that one should now be able to execute prioritized size-based scheduling by simply assigning sockets for small requests to priority queues with high priority (these have low band numbers) and assigning sockets for large requests to low priority queues (these have high band numbers).

It turns out, however that this is not sufficient to get good performance improvement. The reason is somewhat subtle. A lot of the time for servicing a request is made up by connection startup time: specifically, the sending of the SYN ACK by the server. The Linux o.s. sends all such control commands to one particular priority band (band 0). This is not under our control. It is important that when assigning priority bands to sockets we:

1. Never assign any sockets to priority band 0.
2. Make all priority band assignments to bands of *lower* priority than band 0, so that SYN ACKs always have highest priority.

This fix makes connection startup time very low, so that it doesn't dominate the response times of small files.

Note that standard Linux (without the Diffserv patch) sends *all* packets, including SYN ACKs, to the same single priority queue[3]. Thus SYN ACKs have to wait in a long queue, which results in a 200ms startup time for all requests. This 200ms startup time gets added into the response time for short requests, which keeps short requests from doing well.

By keeping the SYN ACKs in their own priority queue, this startup cost can virtually eliminated. This observation was also made very recently in [6]. This fix, *together with* giving short requests priority over long ones, enables the performance of short requests to improve immensely, which is at the heart of our observed improvements. Observe also that giving highest priority to the SYN ACKs does not negatively impact the performance of requests since the SYN ACKs themselves make up only a negligible fraction of the total load.

In Section 7, we described the effect of implementing a "quick fix" which only uses 2 priority bands. All SYN ACKS and all small requests go to the high priority band. All other requests go to the low priority band. As we saw this "quick fix" alone is quite powerful.

## 9 Conclusion

This paper presents a kernel-level implementation of SRPT scheduling of connections in a Web server, and demonstrates that this implementation can significantly improve performance at the Web server.

[3] It is actually possible to have 3 priority queues, but the default is that only one is used.

Mean response time can improve by $200\%$ under low loads and as much as $500\%$ under high loads. Mean slowdown can improve by as much as $700\%$ under high loads. Variance in response time can improve by orders of magnitude. All the while, the SRPT-based implementation hardly penalizes large requests (by "large" requests we mean those that comprise the top $0.5\%$ of the request size distribution). Furthermore these gains are achieved under no loss in byte throughput or request throughput.

This paper also takes a closer look at the Linux kernel from a queueing perspective. Quick fixes to the Linux kernel are proposed which have the effect of limiting the queueing in the kernel, and which benefit many requests without hurting any.

# 10 Limitations of this paper and Future work

The following are some limitations of this paper and plans for future work.

The experiments in this paper involved a specific operating system (Linux), a specific Web server (Flash), and specific workloads (both synthetic and trace-driven). Obvious extensions to this work include extending it to other operating systems, other Web servers, and other workload tests. We do not believe that our results will change for different Web servers or operating systems, but these must be verified.

Our current setup is limited in that we have zero propagation delay. Adding propagation delay may increase the scope of the problem dramatically. For example, once propagation delay is introduced, it is no longer even clear what we mean by the "size" of a request. Should a request for a small file from a client who is far away be considered small or large? If propagation delay is significant, it also might be desirable to consider giving priority to text over images.

Our current setup involves only *static* requests. In future work we plan to expand our technology to schedule cgi-scripts and other *non*-static requests, where size is not necessarily known a priori, but might be able to be guessed, or deduced over time. We expect that the effect of SRPT scheduling might be even more dramatic for cgi-scripts because such requests have much longer running times.

In this paper we have concentrated on reductions in mean response time, mean slowdown, and variance in response time. Another area worth studying is the effect of SRPT-based scheduling on improving the *responsiveness* of a Web server. Web requests are often comprised of text, icons, and images. A client can not make progress until all the text and icons are loaded, but he does not require all the images to be loaded. SRPT-based scheduling would give priority to text and icons (which represent small requests), reducing the time for the Web server to retrieve text and icons by a factor of about 10.

Our current setup considers network bandwidth to be the bottleneck resource and does SRPT-based scheduling of that resource. In a different application where some other resource was the bottleneck (e.g., CPU), it might be desirable to implement SRPT-based scheduling of that resource.

Lastly, at present we only reduce mean delay at the *server*. A future goal is to use SRPT connection-scheduling at proxies. Our long-term goal is to extend our SRPT connection-scheduling technology to routers and switches in the Internet. In this way, the benefit of SRPT scheduling is not just limited to Web servers and other application end-nodes, but rather can help reduce congestion throughout the Internet.

## 11 Acknowledgements

Many people have helped us in this work. Names temporarily omitted for double blind reviewing.

## References

[1] Diffserv-related patches. `http://icawww1.epfl.ch/linux-diffserv/`.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated quality-of-service in Web hosting services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[3] Werner Almesberger. Linux network traffic control — implementation overview. Available at http://lrcwww.epfl.ch/linux-diffserv/.

[4] Werner Almesberger, Jamal Hadi, and Alexey Kuznetsov. Differentiated services on linux. Available at http://lrcwww.epfl.ch/linux-diffserv/.

[5] Baily, Foster, Hoang, Jette, Klingner, Kramer, Macaluso, Messina, Nielsen, Reed, Rudolph, Smith, Tomkins, Towns, and Vildibill. Valuation of ultra-scale computing systems. White Paper, 1999.

[6] Hari Balakrishnan, Venkata Padmanabhan, and Randy Katz. The effects of asymmetry on tcp performance. *ACM Mobile Networks and Applications*, 4(3), 1999.

[7] Paul Barford and Mark E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of SIGMETRICS '98*, pages 151–160, July 1998.

[8] Michael Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[9] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 160–169, May 1996.

[10] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[11] Mark E. Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, October 1999.

[12] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the World Wide Web. In *A Practical Guide To Heavy Tails*, pages 3–26. Chapman & Hall, New York, 1998.

[13] Allen B. Downey. A parallel workload model and its implications for processor allocation. In *Proceedings of High Performance Distributed Computing*, pages 112–123, August 1997.

[14] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of OSDI '96*, October 1996.

[15] Abhijith Halikhedkar, Ajay Uggirala, and Dilip Kumar Tammana. Implemenation of differentiated services in linux (diffspec). Available at http://www.rsl.ukans.edu/ dilip/845/FAGASAP.html.

[16] Internet Town Hall. The internet traffic archives. Available at `http://town.hall.org/Archives/pub-/ITA/`.

[17] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3), 1997.

[18] Mor Harchol-Balter. Task assignment with unknown duration. In *Proceedings of ICDCS '00*, April 2000.

[19] Gordon Irlam. Unix file size survey - 1993. Available at `http://www.base.com/gordoni/ufs93.html`, September 1994.

[20] W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM Sigmetrics*, pages 54–69, 1986.

[21] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceedings of the 1997 USITS*, 1997.

[22] Authors omitted for double-blind reviewing. Analysis of srpt scheduling: Investigating unfairness, (Submitted to Sigmetrics 2001).

[23] Vivek S. Pai, Peter Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of USENIX 1999*, June 1999.

[24] David L. Peterson and David B. Adams. Fractal patterns in DASD I/O traffic. In *CMG Proceedings*, December 1996.

[25] Saravanan Radhakrishnan. Linux – advanced networking overview version 1. Available at http://qos.ittc.ukans.edu/howto/.

[26] J. Roberts and L. Massoulie. Bandwidth sharing and admission control for elastic traffic. In *ITC Specialist Seminar*, 1998.

[27] Linus E. Schrage and Louis W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14:670–684, 1966.

[28] Anees Shaikh, Jennifer Rexford, and Kang G. Shin. Load-sensitive routing of long-lived ip flows. In *Proceedings of SIGCOMM*, September 1999.

## 12 Appendix

The following are results for the same experiments as shown in the paper, except that this time with use the Surge-modified workloads rather than the trace-driven workloads, as explained in Section 4.2.

The Web workload generator we use is an adaptation of the popular Surge Web workload generator [7]. Surge generates HTTP requests that follow the size distribution of empirically-measured request sizes, namely a heavy-tailed distribution with $\alpha$-parameter $1.1$, where most files have size less than 5K bytes, but mean file 11108 bytes. In addition to HTTP request sizes, Surge's stream of HTTP requests also adheres to measurements of the sizes of files stored on the server; the relative popularity of files on the server; the temporal locality present in the request stream; and the timing of request arrivals at the server.

We have modified Surge in several ways. First of all, the Surge workload generator uses a *closed* queueing model. We have modified the code to create an *open* model. This allows us more careful control over the system load, while still adhering to the statistical characteristics of the Surge requests. We use 1000 different file sizes at the Web server, ranging from 76 Bytes to 2 MB, with mean 13727 Bytes.

Second, the Surge system provides an overly-optimistic view of starvation, since it only records response times for those requests which have completed. Thus if there is a large request which never completed (is still sitting in the queue) during the experiment under SRPT, that request would not be recorded by Surge. This is a flaw in many papers which use Surge and yields a lower mean response time for large requests than is actually the case. Since starvation is a central theme in this paper, we have taken extra care in this area. We have modified Surge so that the response times of these large requests which remain in the queue are included in the mean calculation. We measured mean response time for large requests before this change to Surge and then after this change to Surge. We found that the mean response time for large requests was significantly higher after this modification to Surge. All our plots depict these higher starvation numbers.

We discovered several anomalies in Surge's behavior, which we think warrant attention: First, we noticed that the results under Surge are very much influenced by the duration of the experiment. This is due to the fact that the load under Surge changes widely. For example, in one experiment we adjusted the system load to $\rho = 0.8$, but we noticed that the load during the first 5 minutes of the experiment was actually $\rho = 0.6$, and the load during the second 5 minutes of the experiment was actually $\rho = 1.0$. Thus, our results depended on the duration of the experiment. We did not find these fluctuations to be present in the trace.

Another issue was that Surge's Zipf distribution for generating file popularity seemed too severe. In some cases we found that close to $40\%$ of our load was being created by just one medium-size file.

Nonetheless, for completeness, we have repeated all experiments using our modification of Surge.

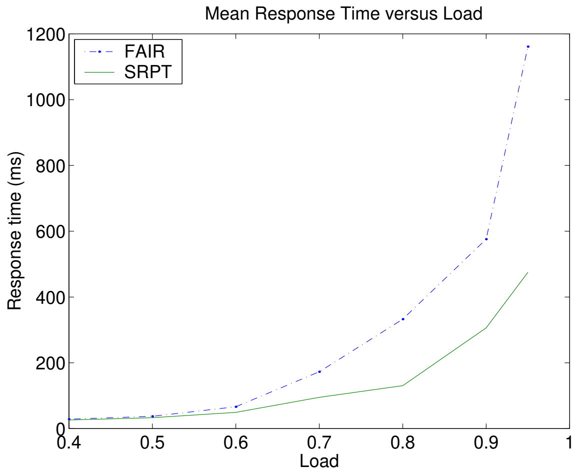



Figure 8: *Mean response time under SRPT scheduling versus traditional FAIR scheduling as a function of system load, under* `Surge`*-modified workload.*

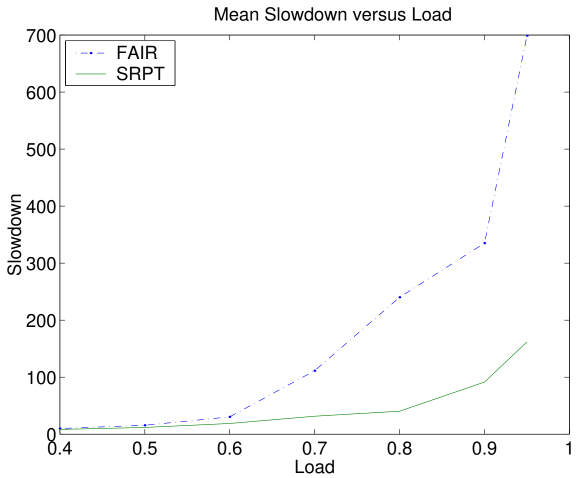
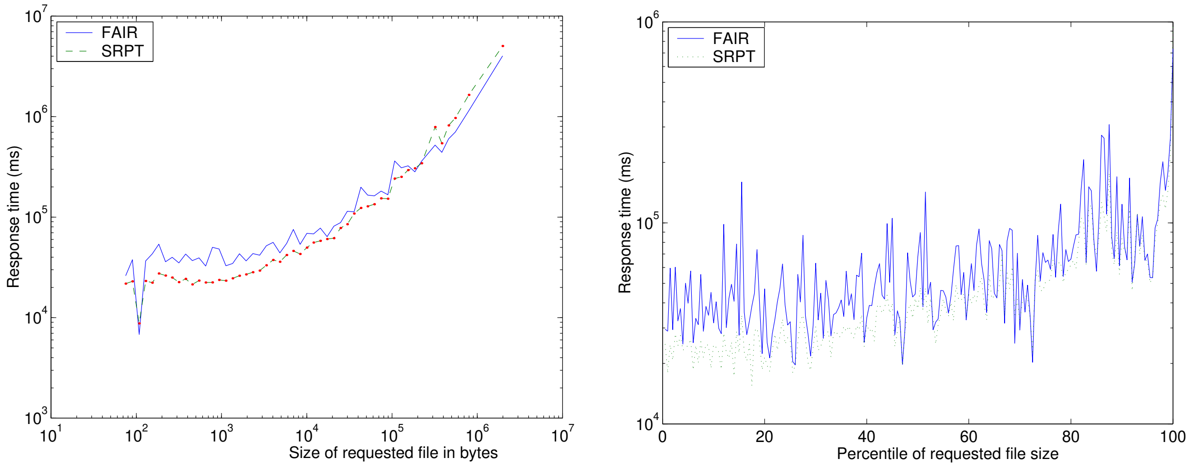
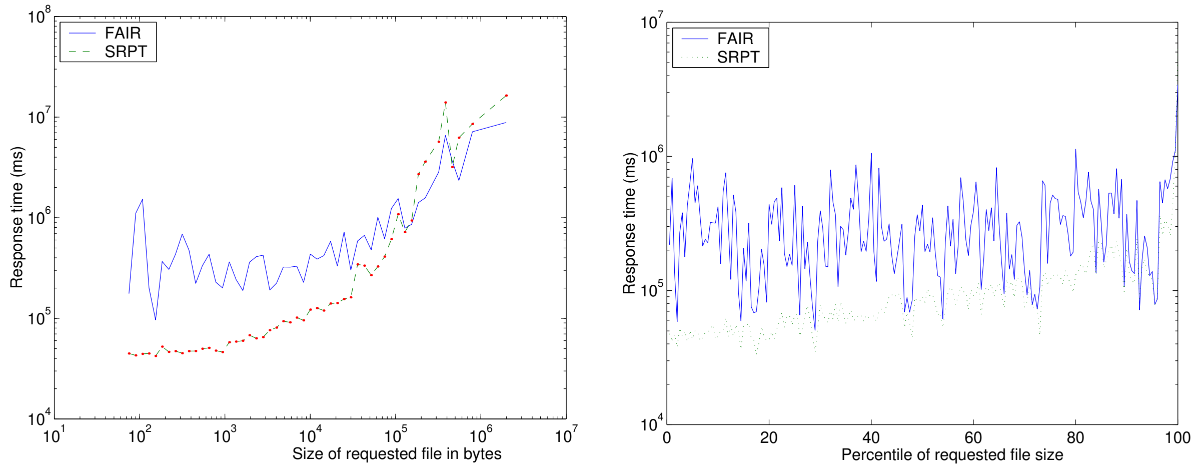


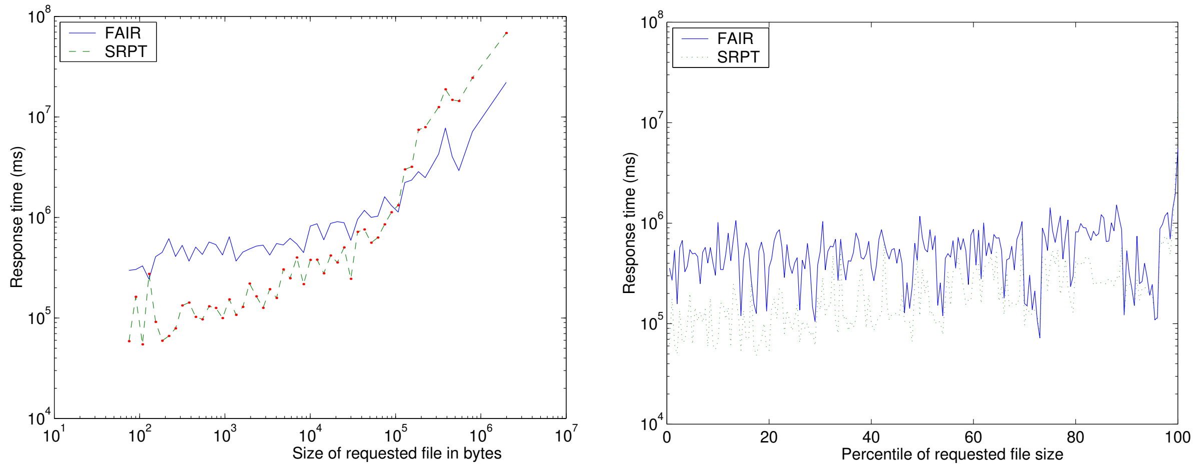

Figure 9: *Mean slowdown under SRPT scheduling versus traditional FAIR scheduling as a function of system load, under* `Surge`*-modified workload.*




(a) **load = .6**

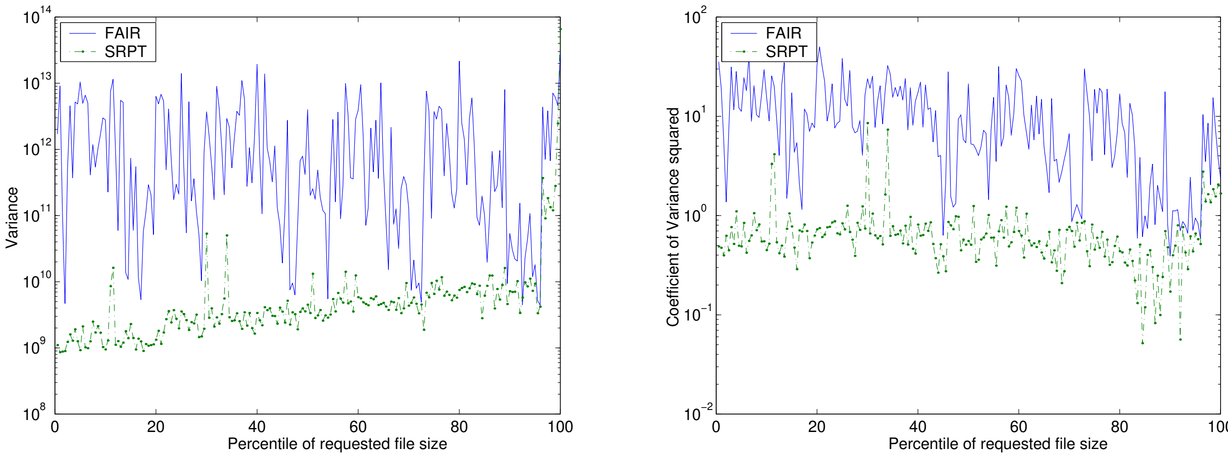



(b) **load = .8**



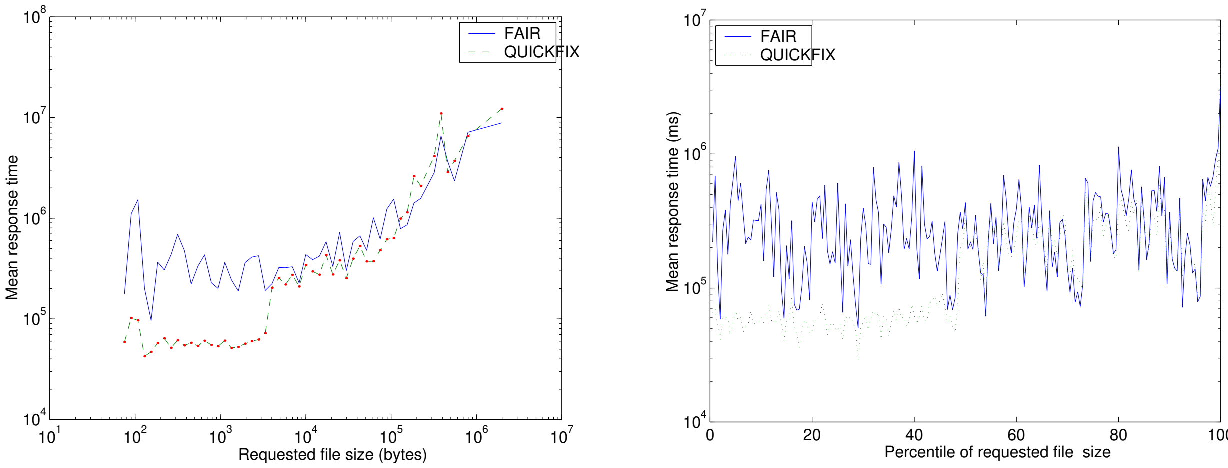

**load = .9**

Figure 10: *Mean response time as a function of request size under* `Surge`*-modified workload, shown for a range of system loads. The left column shows the mean response time as a function of request size. The right column shows the mean response time as a function of the percentile of the request size distribution.*




**(a)** **(b)**

Figure 11: *(a) Variance in response time and (b) Coefficient of variation of response time as a function of the percentile of the request size distribution for SRPT as compared with FAIR scheduling, under* `Surge`*-modified workload with load* = $0.8$.




**(a)** **(b)**

Figure 12: *Mean response time shown (a) as a function of request size and (b) as a function of percentile of request size, for quick-fix scheduling versus traditional FAIR scheduling, under* `Surge`*-modified workload, with load* $0.8$.