Size-based Scheduling to Improve Web Performance

Mor Harchol-Balter* Bianca Schroeder Mukesh Agrawal Nikhil Bansal

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 short, 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 are executed both in a LAN and a WAN environment. We use the Linux operating system and the Apache web server.

Results indicate that SRPT-based scheduling of connections yields significant reductions in delay at the web server. These result in a substantial reduction in mean response time, mean slowdown, and variance in response time for both the LAN and WAN environments.

Significantly, and counter to intuition, the *large requests* are only negligibly penalized or not at all penalized as a result of SRPT-based scheduling.

1 Introduction

A client accessing a busy web server can expect a long wait. This delay is comprised of several components: the propagation delay and transmission delay on the path between the client and the server; delays due to queueing at routers; delays caused by TCP due to loss, congestion, and slow start; and finally the delay at the server itself. The aggregate of these delays, i.e. the time from when the client makes a request until the entire file arrives is defined to be the *response time* of the request.

In this paper we focus on what we can do to improve the delay at the server. Research has shown that in situations where the server is receiving a high rate of requests, the delays at the server make up a significant portion of the response time [8], [7], [28]. More specifically, [8], [7] find that even if the network load is high, the delays at a busy web server can be responsible for more than 80% of the overall response time of a small file, and for 50% of the overall response time of a medium size file.

Measurements [27] suggest that the request stream at most web servers is dominated by *static* requests, of the form "Get me a file." The question of how to service static requests *quickly* is the focus of many companies *e.g.*, Akamai Technologies, and much ongoing research. This paper will focus on *static* requests only.

Our idea is simple. For static 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. 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 call this FAIR scheduling. 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). The expectation is that using SRPT scheduling of requests at the server will reduce the queueing time at the server.

Although it is well-known from queueing theory that SRPT scheduling minimizes queueing time, [36], applications have shied away from using this policy for fear that SRPT "starves" big requests [10, 38, 39, 37]. This intuition is usually true. However, we have a new *theoretical* paper, [30], which proves that in the case of (heavy-tailed) web workloads, this intuition falls apart. In particular, for heavy-tailed workloads, even the largest requests are either *not* penalized at all, or negligibly penalized by SRPT scheduling (see Section 6 for more details). 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. A web server is not a single-resource system. It is not obvious *which* of the web server's resources need to be scheduled. As one would expect, it turns out that scheduling is only important

*This research is funded by Cisco Systems via a grant from the Pittsburgh Digital Greenhouse 00-1 and by NSF-ITR 99-167 ANI-0081396. Equipment was also provided by the Parallel Data Lab.

at the *bottleneck resource*. Frequently this bottleneck resource is the bandwidth on the access link out of the web server. "On a site consisting primarily of *static content*, *network bandwidth* is the most likely source of a performance bottleneck. Even a fairly modest server can completely saturate a T3 connection or 100Mbps Fast Ethernet connection."[25] (also corroborated by [13], [4]). There's another reason why the bottleneck resource tends to be the bandwidth on the access link out of the web site: Access links to web sites (T3, OC3, etc.) cost thousands of dollars per month, whereas CPU is cheap in comparison. Likewise disk utilization remains low since most files end up in the cache. It is important to note that although we concentrate on the case where the network bandwidth is the bottleneck resource, all the ideas in this paper can also be applied to the case where the CPU is the bottleneck — in which case SRPT scheduling is applied to the CPU.

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 the server's 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 requested file. 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. Throughout, we use the Linux OS.

Each experiment is repeated in two ways:

- Under standard Linux (fair-share draining of socket buffers) with an unmodified web server. We call this **FAIR scheduling**.

- Under modified Linux (SRPT-based draining of socket buffers) with the web server modified only to update socket priorities. We call this **SRPT-based scheduling**.

Experiments are executed first in a LAN, so as to isolate the reduction in queueing time at the server. Response time in a LAN is dominated by queueing delay at the server and TCP effects. Experiments are next repeated in a WAN environment. The WAN allows us to incorporate the effects of propagation delay, network loss, and congestion in understanding the full client experience. Response time in a WAN environment represents all these factors, in addition to delay at the server.

In the LAN setting, we experiment with two different web servers: the common Apache server [20], and the Flash web server [33] which is known for speed. Our clients use a request sequence taken from a web trace. All experiments are also repeated using requests generated by a web workload generator (See Section 4.1.2). This request sequence is controlled so that the same experiment can be repeated at many 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 load thus represents the fraction of bandwidth used on the network link out of the web server.

For lack of space, we only include the Apache results in this abstract; the Flash results, which are similar, are in the associated technical report [31].

We obtain the following results in a LAN:

- SRPT-based scheduling decreases mean response time in a LAN by a factor of 3 – 8 for loads greater than 0.5 under Apache.

- SRPT-based scheduling helps small requests a lot, while negligibly penalizing large requests. Under a load of 0.8, 80% of the requests improve by a factor of 10 under SRPT-based scheduling. Only the largest 0.1% of requests suffer an increase in mean response time under SRPT-based scheduling (by a factor of only 1.2).

- The variance in the response time for most requests under SRPT is far lower for *all* requests, in fact two orders of magnitude lower for most requests.

- *SRPT (as compared with FAIR) does not have any effect on the network throughput or the CPU utilization.*

Next we consider a WAN environment, consisting of 6 client machines at various locations within the U.S., feeding 1 sever. For the WAN, we use the Apache web server and again run at different loads.

We obtain the following results in a WAN:

- The improvement in mean response time of SRPT over FAIR under a server load of 0.9 ranged from a factor of 8 (for clients with a Round-trip-time of 100 ms) to a factor of 20 (for clients with an RTT of 20ms). On the other hand there was hardly any improvement in SRPT over FAIR for a server load of 0.5.

- The improvement of SRPT over FAIR in a WAN can actually be greater than in a LAN, for the case of high load at the server.
- Unfairness to large requests is nonexistent in a WAN setting. All request sizes have higher mean response time under FAIR in a WAN environment. We provide theoretical justification for this highly counter-intuitive result in Section 6.

The poor performance of FAIR scheduling throughout encourages us to consider several enhancements to FAIR involving modifications to the Linux kernel. Some of these modifications have been suggested by previous literature and some are new. We find that while some enhancements help somewhat, the don't improve the performance of FAIR to anywhere near the performance of SRPT, in either the LAN setting or the WAN setting.

It is important to realize that this paper is a prototype to illustrate the power of using SRPT-based scheduling. In Section 8, we elaborate on broader applications of SRPT-based scheduling, including its application to other resources, and to non-static requests. We also discuss SRPT applied to web server farms and Internet routers.

2 Previous Work

There has been much literature devoted to improving the response time of web requests. Some of this literature focuses on reducing *network latency*, e.g. by caching requests ([21], [12], [11]) or improving the HTTP protocol ([19], [32]). Other literature works on reducing the *delays at a server*, e.g. by building more efficient HTTP servers ([20], [33]) or improving the server's OS ([18], [5], [26], [29]). Recent studies show that *delays at the server* make up a significant portion of the response time [8], [7]. Our work focuses on reducing delay at the server by using size-based connection scheduling.

In the remainder of this section we discuss only work on *priority-based* or *size-based* scheduling of requests. We first discuss related implementation work and then discuss relevant theoretical results.

Almeida et. al. [1] use both a user-level approach and a kernel-level implementation to prioritizing HTTP requests at a web server. In their experiments, 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 [15]. The authors experiment with connection scheduling at the *application level* only. Via the experimental web server, the authors are able to improve mean response time by a factor of close to 4, but the improvement comes at a price: a drop in throughput by a factor of almost 2.

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 [22, 34, 3, 2].

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

Bender et. al. [10] 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 which does well with respect to max slowdown and mean slowdown.

Roberts and Massoulie [35] consider bandwidth sharing on a link. They suggest that SRPT scheduling may be beneficial in the case of heavy-tailed (Pareto) flow sizes.

The primary theoretical motivation for this paper, comes from our own paper, [30] which will be discussed in Section 6.

3 Implementation of SRPT

In Section 3.1 we explain how socket draining works in standard Linux. In Section 3.2 we describe how to achieve priority queueing in Linux versions 2.2 and above. One problem with size-based queueing 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 3.2.1 describes our solution to this problem. Section 3.3 describes the implementation end at the web server and also deals with the algorithmic issues such as choosing good *priority classes* and setting and updating priorities.

3.1 Default Linux configuration

Figure 1 shows data flow in standard Linux.

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 sep-

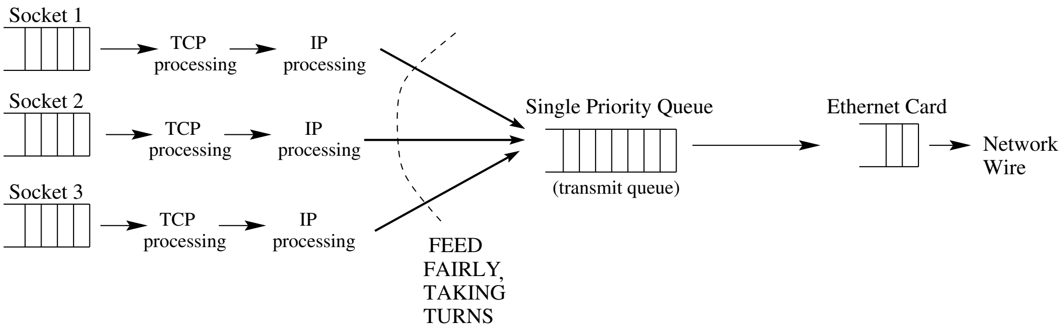


Figure 1: *Data flow in Standard Linux. The important thing to observe is that there is a* single *priority queue into which all connections drain fairly.*

arate. Finally, there is a *single*[1] "priority queue" (*transmit queue*), into which *all* streams feed. In the abstract, these flows take equal turns feeding into the priority queue. Although the Linux kernel does not explicitly enforce fairness, we find that in practice, TCP governs the flows so that they share fairly on short time scales.

This single "priority queue," can get as long as 100 packets. Packets leaving this queue drain into a short Ethernet card queue and out to the network.

3.2 How to achieve priority queueing in Linux

To implement SRPT we need more priority levels. Fortunately, it is relatively easy to achieve up to 16 priority queues (bands), as follows:

First, we build the Linux kernel with support for the user/kernel Netlink Socket, QOS and Fair Queueing, and the Prio Pseudoscheduler. Then we use the `tc`[3] user space tool to switch the device queue from the default 3-band queue to the 16-band prio queue. Further information about the support for differentiated services and various queueing policies in Linux can be found in [22, 34, 3, 2].

Figure 2 shows the flow of data in Linux after the above modification: The processing is the same until the packets reach the priority queue. Instead of a single priority queue (transmit queue), 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 feed fairly into the ith priority queue. The priority queues then feed in a prioritized fashion into the Ethernet Card queue. Priority queue i is only allowed to flow if priority queues 0 through $i-1$ are all empty.

Besides the above modifications to Linux, there is another fix required to make priority queueing effective.

3.2.1 An additional fix – Priority to SYNACKs

An important component of the response time is the connection startup time. In SRPT scheduling, we are careful to separate the small requests from the large ones. However during connection startup, we don't yet know whether the request will be large or small. The packets sent during the connection startup might therefore end up waiting in long queues, making connection startup very costly. For short requests, a long startup time is especially detrimental to response time. It is therefore important that the SYNACK be isolated from other traffic. Linux sends SYNACKs, to priority band 0. It is important that when assigning priority bands to requests that 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 SYNACKs always have highest priority.

Observe that giving highest priority to the SYNACKs does not negatively impact the performance of requests since the SYNACKs themselves make up only a negligible fraction of the total load. Another benefit of giving high priority to SYNACKs is that it reduces their loss probability, which we'll see is sometimes helpful as well.

3.3 Modifications to web server and algorithmic issues in approximating SRPT

The Linux kernel provides mechanisms for prioritized queueing. In our implementation, the Apache web

[1] The queue actually consists of 3 priority queues, a.k.a. bands. By default, however, all packets are queued to the same band.

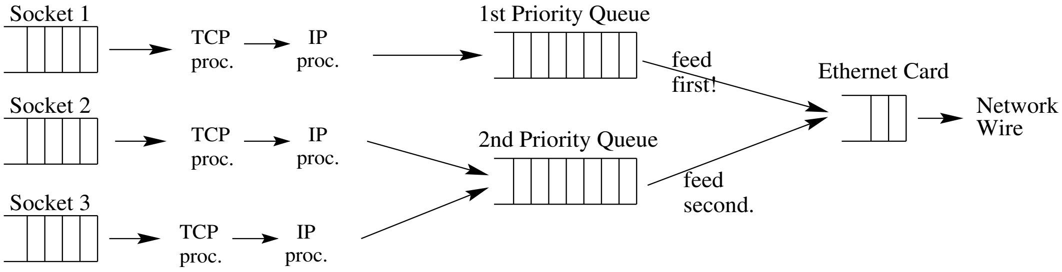


Figure 2: *Flow of data in Linux with priority queueing. It is important to observe that there are several priority queues, and queue i is serviced only if all of queues 0 through $i-1$ are empty.*

server uses these mechanisms to implement the SRPT-based scheduling policy. Specifically, after determining the size of a request, Apache sets the priority of the corresponding socket by calling `setsockopt`. As Apache sends the file, the remaining size of the request decreases. When the remaining size falls below the threshold for the current priority class, Apache updates the socket priority with another call to `setsockopt`.

3.3.1 Implementation Design Choices

Our implementation places the responsibility for prioritizing connections on the web server code. There are two potential problems with this approach. These are the overhead of the system calls to modify priorities, and the need to modify server code.

The issue of system call overhead is mitigated by the limited number of `setsockopt` calls which must be made. In the worst case, we make as many `setsockopt` calls as there are priority classes (6 in our experiments).

The modifications to the server code are minimal. Based on our experience, a programmer familiar with a web server should be able to make the necessary modifications in just a couple of hours.

A clean way to handle the changing of priorities totally within the kernel would be to enhance the `sendfile` system call to set priorities based on the remaining file size. We do not pursue this approach here as neither Apache nor Flash uses `sendfile`.

3.3.2 Size cutoffs

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). We have some *rules-of-thumb* for partitioning the requests into priority classes which apply to the heavy-tailed web workloads. The reader not familiar with heavy-tailed workloads will benefit by first reading Section 6. Denoting the cutoffs by $x_1 < x_2 < \ldots < x_n$:

- The lowest size cutoff x_1 should be such that about 50% of requests have size smaller than x_1. The requests comprise so little total load in a heavy-tailed distribution that there's no point in separating them.
- The highest cutoff x_n needs to be low enough that the largest (approx.) .5% – 1% of the requests have size $> x_n$. This is necessary to prevent the largest requests from starving.
- The middle cutoffs are far less important. Anything remotely close to a logarithmic spacing works well.

In the experiments throughout this paper, we use only 6 priority classes to approximate SRPT. Using more improved performance only slightly.

3.3.3 The final algorithm

Our SRPT-like algorithm is thus as follows:

1. When a request arrives, it is given a socket with priority 0 (highest priority). This is an important detail which allows SYNACKs to travel quickly. This was explained in Section 3.2.1.
2. 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, as shown in the table below.

Priority	Size (Kbytes)
0 (highest)	-
1	$\leq$ 1K
2	1K - 2K
3	2K - 5K
4	5K-20K
5	20K - 50K
6 (lowest)	$>$ 50K

3. 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 LAN setup and experimental results

In Section 4.1 we describe the experimental setup and workload for the LAN experiments. Section 4.2 illustrates the results of the LAN experiments. Section 4.3 proposes some enhancements to improve the performance of FAIR scheduling and describes the results of these enhancements. Lastly Section 4.2.1 illustrates a simplification of the SRPT idea which still yields quite good performance.

4.1 Experimental Setup (LAN)

4.1.1 Architecture

Our experimental architecture involves two machines each with an Intel Pentium III 700 MHz processor and 256 MB RAM, running Linux 2.2.16, and connected by a 10Mb/sec full-duplex Ethernet connection. The Apache web server is running on one of the machines. The other machine hosts the clients which send requests to the web server.

4.1.2 Workload

The clients' requests are generated either via a *web workload generator* (we use a modification of `Surge` [9]) or via *traces*. Throughout this paper, all results shown are for a trace-based workload. We have included in the associated technical report [31] the same set of results for the `Surge` workload.

4.1.3 Traces

The trace-based workload consists of a 1-day trace from the Soccer World Cup 1998, from the Internet Traffic Archive [23]. The 1-day trace contains 4.5 million HTTP requests, virtually all of which are *static*.

An entry in the trace includes: (1) the time the request was received at the server, (2) the size of the request in bytes, (3) the GET line of the request, (4) the error code, as well as other information. In our experiments, we use the trace to specify the time the client makes the request and the size in bytes of the request.

The entire 1 day trace contains requests for approximately 5000 different files. Given the mean file size of 5K, it is clear why all files fit within main memory. This explains why disk is not a bottleneck.

Each experiment was run using a busy hour of the trace (10:00 a.m. to 11:00 a.m.). This hour consisted of about 1 million requests, during which over a *thousand* files are requested.

Some additional statistics about our trace workload: The minimum size file requested is a 41 byte file. The maximum size file requested is about 2 MB. The distribution of the file sizes requested fits a heavy-tailed truncated Pareto distribution (with α-parameter ≈ 1.2). 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.

4.1.4 Generating requests at client machines

In our experiments, we use `sclient` [6] for creating connections at the client machines. The original version of `sclient` makes requests for a certain file in periodic intervals. We modify `sclient` to read in traces and make the requests according to the arrival times and file names given in the trace.

To create a particular load, we simply scale the interarrival times in the trace's request sequence. The scaling factor for the interarrival times is derived both analytically and empirically.

4.1.5 Performance Metrics

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 the time it would require if it were the sole request in the system. Slowdown is also commonly known as *normalized response time* and has been widely used [10, 35, 17, 24].

- *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.

4.2 Main Experimental results (LAN)

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. The same exact set of requests complete under SRPT scheduling and under FAIR scheduling.
- There is no additional CPU overhead involved in SRPT scheduling as compared with FAIR scheduling. Recall that the overhead due to updating priorities of sockets is insignificant, given the small number of priority classes that we use.

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 similar under the two scheduling policies. However for loads > 0.5, the mean response time is a factor of 3 – 8 lower under SRPT scheduling. These results are in agreement with our theoretical predictions in [30].

The results are even more dramatic for mean slowdown. For loads 0.5, the mean slowdown improves by a factor of 4 under SRPT over FAIR. Under a load of 0.9, mean slowdown improves by a factor of 16.

The important question is whether the significant improvements in mean response time come at the price of significant unfairness to large requests. Figure 4 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 4, 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 requests for the very biggest file.

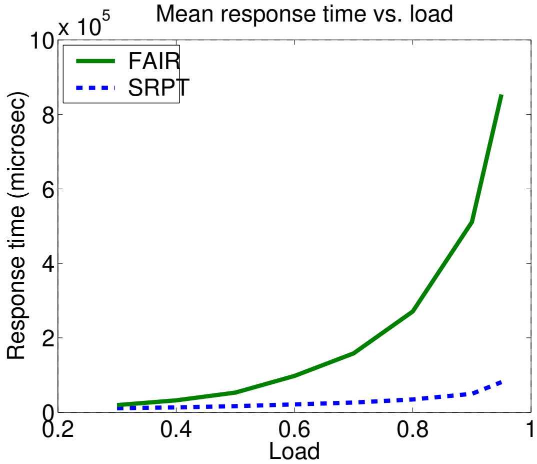
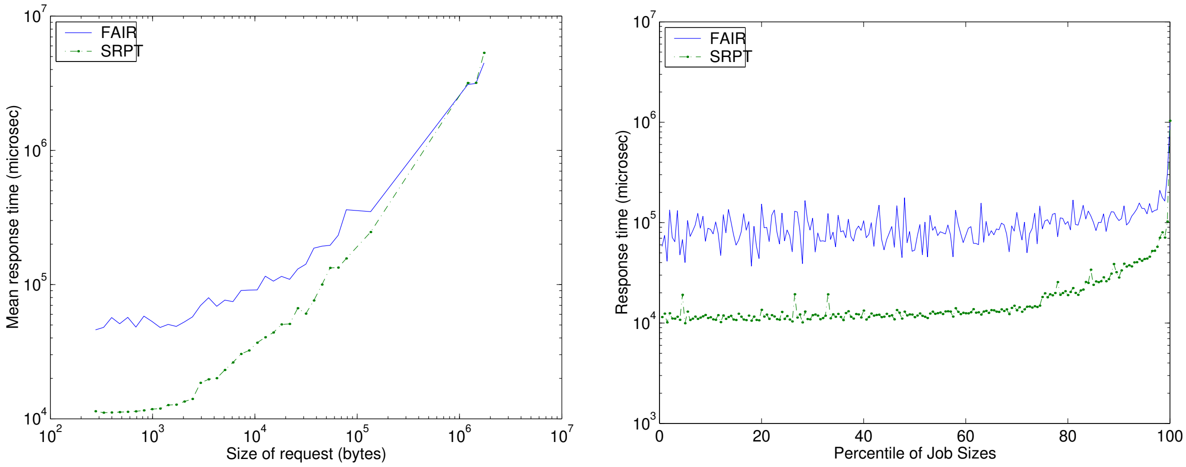


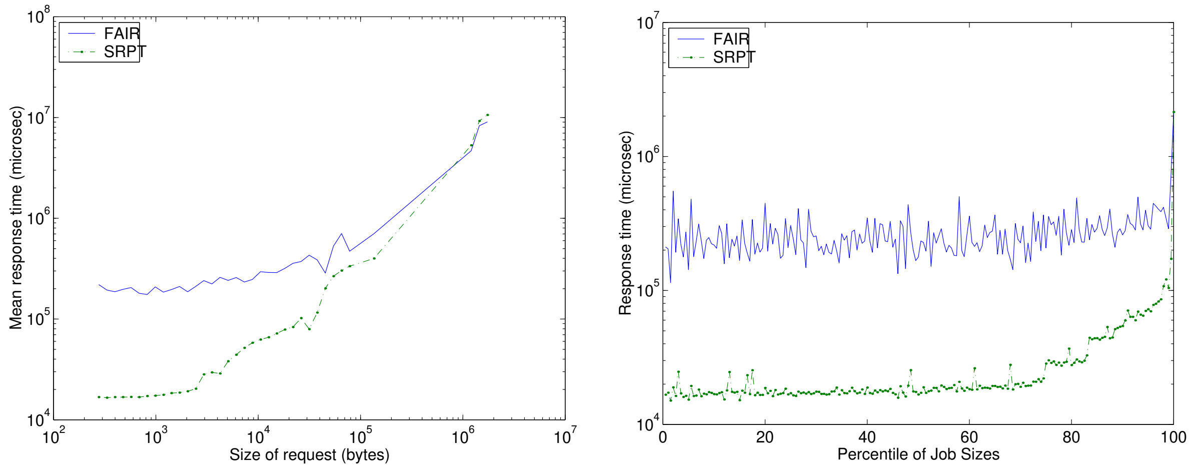
Figure 3: *Mean response time under SRPT versus FAIR as a function of system load, under trace-based workload, in LAN environment.*

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 example, under load of 0.8 (see Figure 4(b)) SRPT scheduling improves the mean response times of small requests by a factor of close to 10, while the mean response time for the largest size request only goes up by a factor of 1.2.

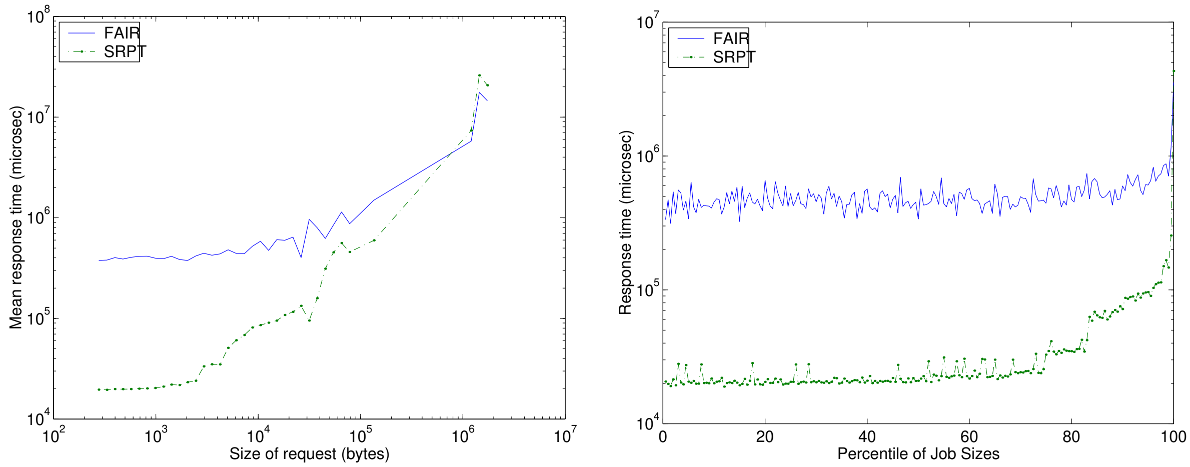
Note that the above plots give equal emphasis to small and large files. As requests for small files are much more frequent, these plots are not a good measure of the improvement offered by SRPT. To fairly assess the improvement, the right column of Figure 4, presents the mean response time as a function 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 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 for the highest load tested though (.95), there are only 500 requests (out of the 1 million requests) which complete later under SRPT as compared with FAIR. These requests are so large however, that the effect on their slowdown is negligible.



(a)**load** = .6



(b)**load** = .8



(c)**load** = .9

Figure 4: *Mean response time as a function of request size under trace-based workload, shown for a range of system loads, in a LAN. 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.*

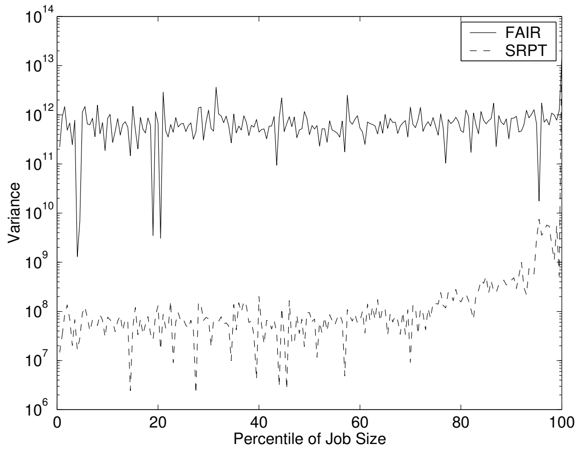


Figure 5: *Variance in response time as a function of the percentile of the request size distribution for SRPT as compared with FAIR, under trace-based workload with load* = 0.8*, in a LAN.*

Figure 5 shows the variance in response time for each request size as a function of the percentile of the request size distribution, for load equal to 0.8. The improvement under SRPT with respect to variance in response time is 2 – 4 orders of magnitude for the 99.5% smallest files. The improvement with respect to the squared coefficient of variation (variance/mean^2) is about 30.

4.2.1 Parameter Sensitivity in SRPT (LAN)

To evaluate the importance of choosing precise cutoffs, we evaluate SRPT with only two priority classes. We define *small* requests as the smallest 50% of requests and *large* requests as the largest 50% of requests (note, this is not the same thing as equalizing load) The cutoff falls at 1K. We find that this simple algorithm results in a factor of 2.5 improvement in mean response time and a factor of 5 improvement in mean slowdown over FAIR.

4.3 Enhancements to FAIR (LAN)

At this point it is natural to wonder why the FAIR policy performs so poorly. The obvious reason is that the time-sharing behavior of FAIR causes all requests to be delayed, which leads to high response times and high numbers of requests in the system. By contrast, SRPT works to minimize the number of requests in the system, and thus the mean response time as well.

Despite the above argument, one can't help but wonder whether Linux or Apache/Flash itself is causing FAIR to perform especially badly.

To answer this question, we instrumented the web server's kernel to provide statistics including: the occupancy of the SYN and ACK (listen) queues, the number of incoming SYNs dropped due to the SYN queue being full, the number of times a client's acknowledgement of a SYNACK was discarded due to the ACK queue being full, and the number of outgoing SYNACKs dropped inside the kernel.

Under the newly instrumented kernel, we reran all the LAN experiments. Below we discuss our findings just for the case of load 0.9. For this case the response time was 452ms under FAIR and 38ms under SRPT. Our measurements indicate that under FAIR, a significant fraction (5%-10%) of connections suffered long delays due to loss at the server. Under SRPT, this effect is virtually non-existent.

Effect of length of transmit queue

Consider Figure 1 which shows flow of control in standard Linux (FAIR). Observe that all socket buffers drain into the same single priority queue. This queue may grow long. 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 120 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 10-20 ms.

In our first experiment, we shortened the length of the transmit queue. This resulted in an *increase* in mean response time — from 452ms to 629ms under FAIR. The problem is that by shortening the length of the transmit queue, we increase the loss.

We next tried to lengthen the transmit queue, increasing it from 100 to 500, and then to 700. This helped a little — reducing mean response time from 452ms to 342ms. The reason was a reduction in loss. Still, performance was nowhere near that of SRPT – 38ms.

Effect of length of SYN and ACK queues

We observe that in the LAN experiments neither the SYN queue nor the ACK queue ever fills to capacity. Therefore increasing its length has no effect.

Effect of giving priority to SYNACKs

Recall in Section 3.2.1 we showed that giving priority to SYNACKs was an important component of implementing SRPT. We therefore decided to try the same idea for FAIR. We found that when SYNACKs were given priority the mean response time dropped from 452ms to only 265ms – a decent improvement, but still nowhere close to the performance of SRPT.

Lastly we combined all 3 enhancements to FAIR. The performance remained at 265ms for FAIR.

5 WAN setup and Experimental Results

In Section 5.1 we describe the setup for the WAN experiments. In Section 5.2 we describe the results of the WAN experiments. In Section 5.3 we consider the effect of several enhancements to FAIR scheduling in the WAN environment.

5.1 Experimental setup (WAN)

We use the same server machine as for the LAN experiment. This time we have 6 client machines located throughout the Internet. The clients generate the requests in the same way as before based on a trace. Again each experiment again spans 1 hour (about 1 million requests).

The clients are located at various distances from the server (indicated by round trip times, RTT) and have varying available bandwidth, as shown in the table below.

Location	Avg. RTT	Avail. Bndwth
IBM, New York	20ms	8Mbps
Univ. Berkeley	55ms	3Mbps
UK	90-100ms	1Mbps
Univ. Virginia	25ms	2Mbps
Univ. Michigan	20ms	5Mbps
Boston Univ.	22ms	1Mbps

5.2 Experimental results for the WAN setup

Figure 6 shows the mean response time (in ms) as a function of load for each of the six hosts. This figure show that the improvement in mean response time of SRPT over FAIR is a factor of 8–20 for high load (0.9) and only about 1.1 for lower load (0.5).

Figure 7(a) and 7(b) shows the mean response time of a request as function of the percentile the request size at a load of 0.8 for the hosts at IBM and UK respectively. It's not clear from looking at the figures whether there is any starvation. It turns out that *all* request sizes have higher mean response time under FAIR, as compared with SRPT. For the largest file, the mean response time is almost the same under SRPT and FAIR.

We also measured the variance in response time (graph omitted for lack of space) in the WAN environment for a load of 0.8. While the variance for FAIR stayed the same under the LAN and WAN environments, the variance for SRPT increased somewhat in the WAN environment due to losses. Still, however, the variance in response time under SRPT remains over an order of magnitude below that in FAIR, for a load of 0.8.

We make the following observations:

Observation 1 *The improvement of SRPT over FAIR in mean response time is greater at higher loads.*

For example in Figure 6 the mean response time for the IBM host under SRPT improves over FAIR by a factor of 20,3,1.5 and 1.1 at loads 0.9,0.8,0.7 and 0.5 respectively. *Explanation:* We already saw in a LAN that under higher load, the difference between SRPT and FAIR is higher. This is coupled with the fact that at higher load the queueing delay at the server makes up a larger component of response time.

Observation 2 *The improvement of SRPT over FAIR is less for far away clients.*

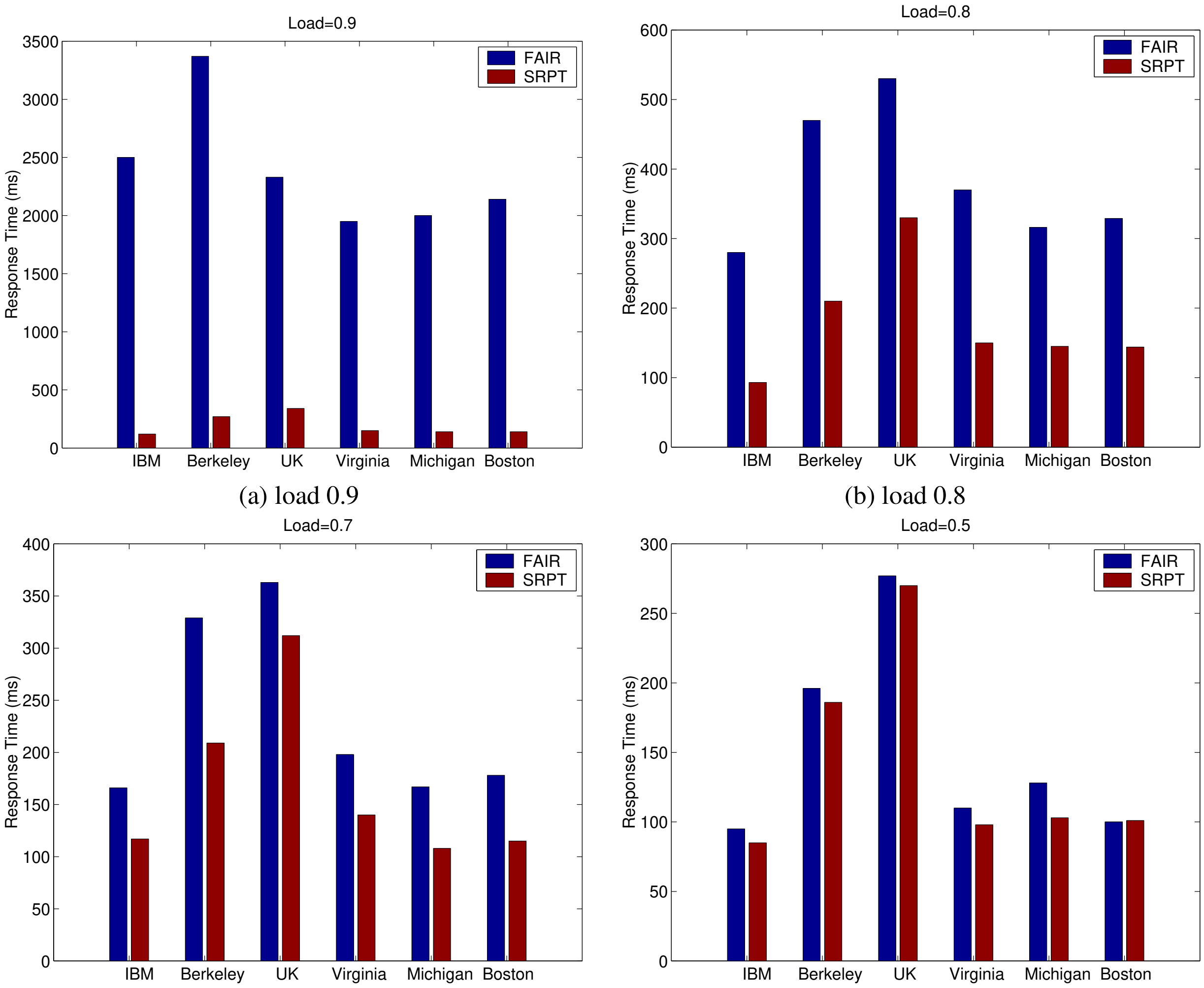
For example in Figure 6, at load 0.8, the mean response time for the IBM host (RTT 20ms) improves by a factor of about 3 under SRPT over FAIR, whereas there is only a factor 1.6 improvement for the far away host at UK (RTT 90 ms). *Explanation:* The delays caused by *propagation* and *Internet congestion* mitigate the effect of the queueing delay on total response time.

Observation 3 *The improvement of SRPT over FAIR in a WAN environment can actually be greater than in a LAN environment, for the case of high load at the server.*

For example in Figure 6 the mean response time for the host at IBM is 2500 ms under FAIR scheduling, vs. 125 ms under SRPT scheduling, hence about 20 times better. However, in the LAN setup the mean response time improved by a factor of about 12 at load 0.9 (See Figure 3).

Explanation: This surprising observation is due to effects not yet considered: *loss*, and the effect of loss on *TCP*.

Observe that the mean response times under FAIR are very high (at least 2500 ms) at load 0.9. This suggests that some loss is occurring during the early



(a) load 0.9

(b) load 0.8

(c) load 0.7

(d) load 0.5

Figure 6: *Mean response time under SRPT versus FAIR in a WAN under load (a) 0.9, (b) 0.8, (c) 0.7, and (d) 0.5.*

parts of the connections (when the retransmit timeout penalties are high). Our measurements show the server under FAIR is in fact dropping about 7% of the SYN – connection requests from the IBM client, as compared with only 0.2% under SRPT.

The reason that SYNs are being dropped is that the SYN queue (which stores SYNs at the server) under FAIR is almost always full. For a SYN to be removed from the SYN queue, requires that a SYNACK (acknowledgement for the SYN) be sent by the server and a final ACK received from the client. The problem is that the SYNACKs are delayed in leaving the server under FAIR (they wait up to 120 ms in the transmit queue), causing the SYN to sit in the SYN queue an unduly long time.

Observation 4 *The mean response times under SRPT are close to optimal even under high loads.*

For example, for the host at Berkeley (RTT 55ms), the mean response times are 186, 209, 210 and 270 ms at loads 0.5,0.7,0.8 and 0.9 respectively under SRPT scheduling. Observe that these are quite close to 170ms, which is the optimal mean response time (i.e. when the load at the server is close to 0) for this host.

Observation 5 *While the penalty of SRPT to large requests is almost absent in the LAN setting (see Figure 4), we observe that it is even less of an issue in the WAN environment.*

As explained above now *all* request sizes have higher mean response time under FAIR, as compared with SRPT. *Explanation:* The reason is simply that the propagation delay in the case of a WAN mitigates the effect of the queueing delay (in particular the difference between the queueing delay under SRPT and that under FAIR).

5.3 Enhancements to FAIR (WAN)

In Section 4.3 we considered several enhancements to FAIR scheduling. For completeness, we again tried these enhancements in the WAN setting. We find that increasing the length of both the SYN and ACK queues simultaneously improves upon the response time of FAIR by almost a factor of 2. This corroborates our explanation of Observation 3. Prioritizing SYNACKs did not have significant effect. Increasing the length of the transmit queue reduced performance. Note that SRPT improves upon the performance of the best FAIR configuration by a factor of 5-10 (depending on the client host) under load 0.9.

6 How can *every* request prefer SRPT to FAIR in expectation? Theoretical Explanation

It has 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.

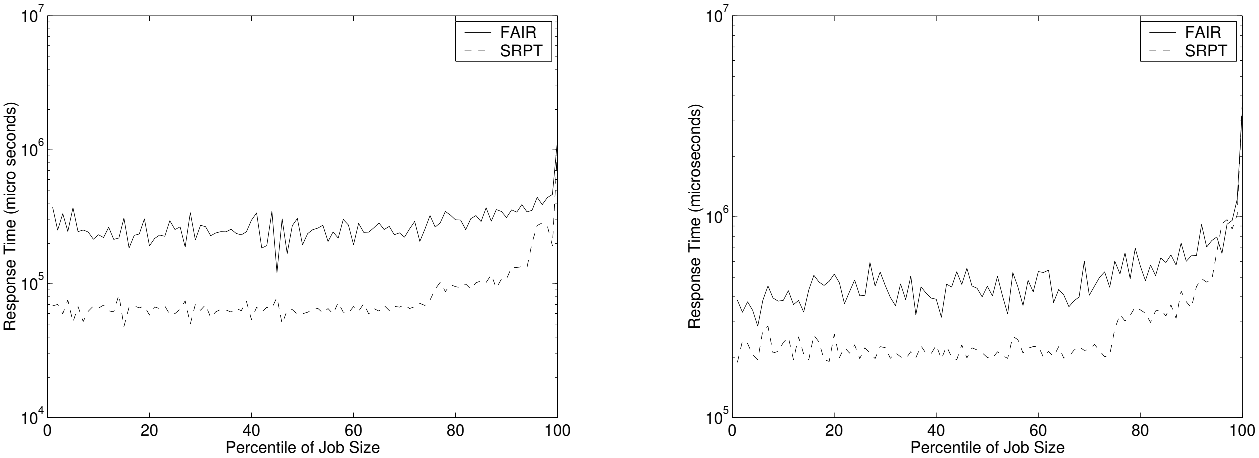
2. The property that a tiny fraction (usually $< 1\%$) of the very longest requests comprise over half of the total load. We refer to this important property as the **heavy-tailed property**.

The lower the parameter α, the more variable the distribution, and the more pronounced is the heavy-tailed property, *i.e.* the smaller the fraction of long requests that comprise half the load.

Request sizes are well-known to follow a heavy-tailed distribution [14, 16]. Our traces also have strong heavy-tailed properties. (In our trace the largest $< 3\%$ of the requests make up $> 50\%$ of the total load.)

Consider a workload where request sizes exhibit the *heavy-tailed property*. Now 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. 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



(a) IBM clients

(b) UK clients

Figure 7: *Response time as a percentile of request size under SRPT scheduling versus traditional FAIR scheduling at load 0.8, measured for (a) the IBM host and (b) the UK host.*

(i.e. the largest possible request). It turns out that, provided the load is not too close to 1, the request in the 100%-tile will quickly see an idle period, during which it can run. As soon as the request gets a chance to run, it will quickly become a request in the 99.5%-tile, at which time it will clearly prefer SRPT. For a formalization of the above argument, we refer the reader to [30].

7 Conclusion

This paper demonstrates that the delay at a busy server can be greatly reduced by SRPT-based scheduling of requests at the server's outgoing link. We show further that the reduction in server delay often results in a reduction in the client-perceived response time.

Our SRPT-based scheduling algorithm reduces mean response time in a LAN setting significantly under high server loads, over the standard FAIR scheduling algorithm. In a WAN setting the improvement is similar for very high server loads, but is less significant at moderate loads.

Surprisingly, this improvement comes at no cost to large requests, which are hardly penalized, or not at all penalized. Furthermore these gains are achieved under no loss in byte throughput or request throughput.

8 Limitations and Future work

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. Determining the size (processing requirement) of non-static requests is an important open problem, but much progress is being made on better predicting the size of dynamic requests, or deducing them over time.

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

Although we evaluate SRPT and FAIR across many server loads, we do not in this paper consider the case of overload. This is an extremely difficult problem both analytically and especially experimentally. Our preliminary results show that in the case of *transient overload* SRPT outperforms FAIR across a long list of metrics, including mean response time, throughput, server losses, etc.

Our SRPT solution can also be applied to server farms. In this scenario the bottleneck moves from the outgoing link at each server to the access link for the server farm — thus scheduling needs to be applied at the access link. To achieve this, servers would mark packets to designate their priority. These priorities would be enforced by the router at the access link.

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.

References

[1] 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.

[2] W. Almesberger, J. Hadi, and A. Kuznetsov. Differentiated services on linux. Available at http://lrcwww.epfl.ch/linux-diffserv/.

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

[4] Bruce Maggs at Akamai. Personal communication., 2001.

[5] G. Banga, P. Druschel, and J. Mogul. Better operating system features for faster network servers. In *Proc. Workshop on Internet Server Performance*, June 1998.

[6] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server under realistic loads. *World Wide Web*, 2(1-2):69–83, 1999.

[7] Paul Barford and M. E. Crovella. Measuring web performance in the wide area. *Performance Evaluation Review – Special Issue on Network Traffic Measurement and Workload Characterization*, August 1999.

[8] Paul Barford and Mark Crovella. Critical path analysis of tcp transactions. In *SIGCOMM*, 2000.

[9] 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.

[10] 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.

[11] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abdelsalam Heddaya, and Sulaiman A. Mirdad. Application-level document caching in the internet. In *Proceedings of the Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)*, June 1995.

[12] H. Braun and K. Claffy. Web traffic characterization: an assessment of the impact of caching documents from NCSA's Web server. In *Proceedings of the Second International WWW Conference*, 1994.

[13] Adrian Cockcroft. Watching your web server. The Unix Insider at http://www.unixinsider.com, April 1996.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] Fielding, Gettys, Mogul, Frystyk, and Berners-lee. DNS support for load balancing. RFC 2068, April 1997.

[20] The Apache Group. Apache web server. `http://www.apache.org`.

[21] James Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings of HotOS '94*, May 1994.

[22] A. Halikhedkar, Ajay Uggirala, and D.K. Tammana. Implemenation of differentiated services in linux (diffspec). Available at http://www.rsl.ukans.edu/ dilip/845/FAGASAP.html.

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

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

[25] Microsoft TechNet Insights and Answers for IT Professionals. The arts and science of web server tuning with internet information services 5.0. http://www.microsoft.com/technet/, 2001.

[26] M. Kaashoek, D. Engler, D. Wallach, and G. Ganger. Server operating systems. In *SIGOPS European Workshop*, September 1996.

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

[28] Evangelos Markatos. Main memory caching of Web documents. In *Proceedings of the Fifth Interntional Conference on the WWW*, 1996.

[29] J. Mogul. Operating systems support for busy internet servers. Technical Report WRL-Technical-Note-49, Compaq Western Research Lab, May 1995.

[30] Authors omitted for purpose of double-blind reviewing. Analysis of SRPT scheduling: Investigating unfairness. In *To appear in Proceedings of* Sigmetrics '01.

[31] Authors omitted for purpose of double-blind reviewing. Implementation of SRPT scheduling in web servers. Technical Report XXX-CS-00-170, 2000.

[32] V. N. Padmanabhan and J. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28:25–35, December 1995.

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

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

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

[36] 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.

[37] A. Silberschatz and P. Galvin. *Operating System Concepts, 5th Edition*. John Wiley & Sons, 1998.

[38] W. Stallings. *Operating Systems, 2nd Edition*. Prentice Hall, 1995.

[39] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.