SRPT Scheduling for Web Servers

Mor Harchol-Balter, Nikhil Bansal, Bianca Schroeder, and Mukesh Agrawal

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213
{harchol, nikhil, bianca, mukesh}@cs.cmu.edu

Abstract. This note briefly summarizes some results from two papers: [4] and [23]. These papers pose the following question:

> *Is it possible to reduce the expected response time of* every *request at a web server, simply by changing the order in which we schedule the requests?*

In [4] we approach this question analytically via an M/G/1 queue. In [23] we approach the same question via implementation involving an Apache web server running on Linux.

1 Introduction

Motivation and goals

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.

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 [6], [5], [32].

Our work will focus on *static* requests only of the form "Get me a file." Measurements [31] suggest that the request stream at most web servers is dominated by *static* requests. The question of how to service static requests *quickly* is the focus of many companies *e.g.*, Akamai Technologies, and much ongoing research.

Our idea

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, 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 (a.k.a. Processor-Sharing 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.

The controversy

It has long been known that SRPT has the lowest mean response time of any scheduling policy, for any arrival sequence and job sizes [41, 46]. Despite this fact, applications have shied away from using this policy for fear that SRPT "starves" big requests [9, 47, 48, 45]. It is often stated that the huge average performance improvements of SRPT over other policies stem from the fact that SRPT unfairly penalizes the large jobs in order to help the small jobs. It is often thought that the performance of small jobs cannot be improved without hurting the large jobs and thus large jobs suffer unfairly under SRPT.

2 Analysis of SRPT based on [4]

Relevant previous work

It has long been known that SRPT minimizes mean response time [41, 46]. Rajaraman et al. showed further that the mean slowdown under SRPT is at most twice optimal, for any job sequence [19].

Schrage and Miller first derived the expressions for the response times in an M/G/1/SRPT queue [42]. This was further generalized by Pechinkin *et al.* to disciplines where the remaining times are divided into intervals [36]. The steady-state appearance of the M/G/1/SRPT queue was obtained by Schassberger [40]. The mean response time for a job of size x in an M/G/1/SRPT server is given below:

$$E[T(x)]_{SRPT} = \frac{\lambda(\int_0^x t^2 f(t)dt + x^2(1-F(x)))}{2(1-\lambda\int_0^x tf(t)dt)^2} + \int_0^x \frac{dt}{1-\lambda\int_0^t yf(y)dy}$$

where λ is the average arrival rate and $f(t)$ is the p.d.f. of the job size distribution.

The above formula is difficult to evaluate numerically, due to its complex form (many nested integrals). Hence, the comparison of SRPT to other policies was long neglected. More recently, SRPT has been compared with other policies by plotting the mean response times for specific job size distributions under specific loads [39, 37, 43]. These include a 7-year long study at University of Aachen under Schreiber et. al. These results are all plots for *specific* job size distributions and loads. Hence it is not clear whether the conclusions based on these plots hold more generally.

It is often cited that SRPT may lead to *starvation* of large jobs [8, 47, 48, 45]. Usually, examples of adversarial request sequences are given to justify this. However, such worst case examples do not reflect the behavior of SRPT on average. The term "starvation" is also used by people to indicate the *unfairness* of SRPT's treatment of long jobs. The argument given is that if a scheduling policy reduces the response time of small jobs, then the response times for the large jobs would have to increase considerably in accordance with conservation laws. This argument is true for scheduling policies which *do not* make use of size, see the famous Kleinrock Conservation Law [28], [29, Page 197].

Very little has been done to evaluate the problem of unfairness analytically. Recently, Bender et al. consider the metric *max slowdown* of a job, as indication of unfairness [8]. They show with an example that SRPT can have an arbitrarily large *max slowdown*. However, *max slowdown* may not be the best metric for unfairness. One large job may have an exceptionally long response time in some case, but it might do well most of the time. A more relevant metric is the *max mean slowdown*.

The question of how heavy-tailed workloads might affect SRPT's performance has not been examined.

Our model

Throughout paper [4] we assume an M/G/1 queue where G is assumed to be a continuous distribution with finite mean and variance.

It turns out that the job size distribution[1] is important with respect to evaluating SRPT. We will therefore assume a general job size distribution. We will also concentrate on the special case of distributions with the **heavy-tailed property** (**HT property**). We define the HT property to say that *the largest 1% of the jobs comprise at least half the load*. This HT property appears in many recent measurements of computing system workloads, including both sequential jobs and parallel jobs [30, 24, 13, 26, 38, 44]. In particular the sizes of *web files* requested and the sizes of web files stored have been shown to follow a Pareto distribution which possesses the HT property [7, 14, 16].

Some results from [4]

In [4], we prove the following results, among others:

- Although it is well-known that SRPT scheduling optimizes mean response time, it is not known how SRPT compares with Processor-Sharing scheduling (a.k.a. FAIR scheduling) with respect to mean slowdown. We prove that SRPT scheduling also outperforms Processor-Sharing (PS) scheduling with respect to mean slowdown for all job size distributions.

[1] Note: By "the size of a job" we mean the service requirement of the request. In the case of static web requests, this is proportional to the number of bytes in the request.

- Given that SRPT improves performance over PS both with respect to mean response time and mean slowdown, we next investigate the magnitude of the improvement. We prove that for all job size distributions with the HT property the improvement is very significant under high loads. For example, for load 0.9, SRPT improves over PS with respect to mean slowdown by a factor of at least 4 for all distributions with the HT property. As the load approaches 1, we find that SRPT improves over PS with respect to mean slowdown by a factor of 100 for all distributions with the HT property. In general we prove that for *all* job size distributions as the load approaches one, the mean response time under SRPT improves upon the mean response time under PS by at least a factor of 2 and likewise for mean slowdown.
- The performance improvement of SRPT over PS does *not* usually come at the expense of the large jobs. In fact, we observe via example that for many job size distributions with the HT property every single job, including a job of the maximum possible size, prefers SRPT to PS (unless the load is extremely close to 1).
- While the above result does not hold at all loads, we prove that no matter what the load, at least 99% of the jobs have a lower expected response time under SRPT than under PS, for all job size distributions with the HT property. In fact, these 99% of the jobs do significantly better. We show that these jobs have an average slowdown of at most 4, at any load $\rho < 1$, whereas their performance could be arbitrarily bad under PS as the load approaches 1. Similar, but weaker results are shown for general distributions.
- While the previous result is concerned only with 99% of the jobs, we also prove upper bounds on how much worse any job could fare under SRPT as opposed to PS for general distributions. Our bounds show that jobs never do too much worse under SRPT than under PS. For example, for all job size distributions, the expected response time under SRPT for any job is never more than 3 times that under PS, when the load is 0.8, and never more than 5.5 times that under PS when the load is 0.9. In fact, if the load is less than half, then *for every job size distribution, each job has a lower expected response time and slowdown under SRPT than under PS.*
- The above results show an upper bound on how much worse a job could fare under SRPT as opposed to PS for general job size distributions. We likewise prove lower bounds on the performance of SRPT as compared with PS for general job size distributions.

3 Implementation of SRPT based on [23]

Relevant previous systems 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], [11], [10]) or improving the HTTP protocol ([18], [34]). Other literature works on reducing the *delays at a server*, e.g. by building more

efficient HTTP servers ([20], [35]) or improving the server's OS ([17], [3], [27], [33]).

The solution we propose is different from the above in that we only want to change the order in which requests are scheduled. In the remainder of this section we discuss only work on *priority-based* or *size-based* scheduling of requests.

Almeida et. al. [1] use both a user-level approach and a kernel-level implementation to prioritizing HTTP requests at a web server. The *user-level* approach in [1] involves modifying the Apache web server to include a Scheduler process which determines the order in which requests are fed to the web server. The *kernel-level* approach in [1] simply sets 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. For example, 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 which deals specifically with SRPT scheduling at web servers is our own earlier work [15]. This implementation does not involve any modification of the kernel. We experiment with connection scheduling at the *application level* only. We 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 problem 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, we are forced to limit the throughput of requests.

Our approach

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 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 [12], [2]). 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 algorithm 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. 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.

Figure 1 shows data flow in standard Linux, which employs FAIR scheduling. Data streaming into each socket buffer is encapsulated into packets which obtain TCP headers and IP headers. Finally, there is a *single*[2] "priority queue" (*transmit queue*), into which *all* streams feed. This single "priority queue," can get as long as 100 packets.

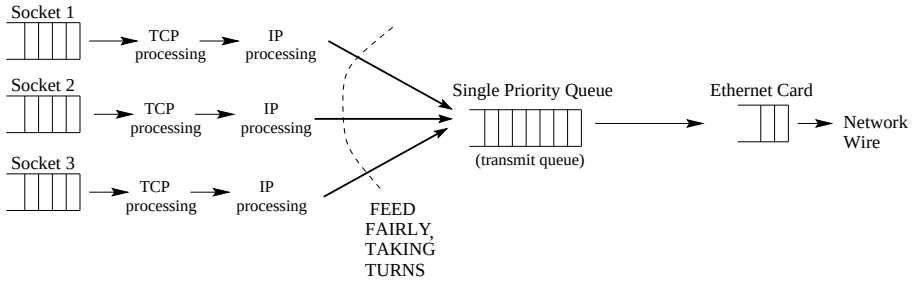

Fig. 1. *Data flow in Standard Linux — FAIR scheduling.*

Figure 2 shows the flow of data in Linux after our modifications: Instead of a single priority queue (transmit queue), there are multiple priority queues. Priority queue i is only allowed to flow if priority queues 0 through $i-1$ are all empty. We used 6 priority queues in our experiments.

After modifying the Linux kernel, we next had to modify the Apache web server to assign priorities in accordance with SRPT. Our modified Apache determines the size of a request and then 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.

Lastly, we had to come up with an algorithm for partitioning the requests into priority classes which work well with the heavy-tailed web workload.

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

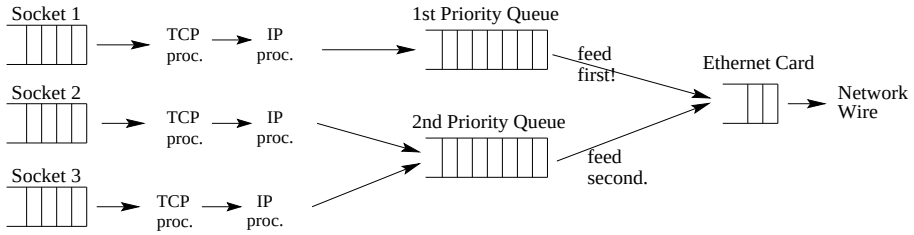

Fig. 2. *Flow of data in Linux with priority queueing (2 priorities shown)*

The combination of (i) the modifications to Linux, (ii) the modifications to the Apache web server, and (iii) the priority algorithm allows us to implement SRPT scheduling. Details on each of these three components are provided in [23].

A very simple experimental architecture is used to run our tests. It involves only 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 run on one of the machines. The other machine (referred to as the "client machine") hosts 200 or so (simulated) client entities which send requests to the web server.

The client's requests are taken from a 1-day trace from the Soccer World Cup 1998, from the Internet Traffic Archive [22]. The 1-day trace contains 4.5 million HTTP requests, virtually all of which are *static*. The trace exhibits a strong heavy-tailed property with the largest $< 3\%$ of the requests making up $> 50\%$ of the total load.

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 example if the requests require 8Mb/sec of bandwidth, and the available bandwidth on the link if 10Mb/sec, then the network load is 0.8).

Some results from [23]

Our experiments yield the following results:

- SRPT-based scheduling decreases mean response time in our LAN setup by a factor of $3 - 8$ for loads greater than 0.5.
- 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 is far lower under SRPT as compared with FAIR, in fact two orders of magnitude lower for most requests.

– *There is no negative effect on byte throughput, request throughput, or CPU utilization from using SRPT as compared with FAIR.*

For more details see [23].

4 Conclusion

We have shown both analytically and experimentally that SRPT scheduling of requests is very powerful under workloads with a heavy-tail property, such as web workloads. Under such workloads, 99% of requests see significant improvement in mean response time under SRPT scheduling as compared with the traditionally-used FAIR scheduling. Furthermore, even the very largest requests have lower expected response time under SRPT than under FAIR scheduling in theory. Experimentally, the largest requests may perform negligibly worse under SRPT scheduling as compared with FAIR scheduling. We believe this is simply due to the coarseness of the implementation.

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. Bruce Maggs at Akamai. Personal communication., 2001.
3. G. Banga, P. Druschel, and J. Mogul. Better operating system features for faster network servers. In *Proc. Workshop on Internet Server Performance*, June 1998.
4. Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *Proceeding of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '01)*, June 2001.
5. 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.
6. Paul Barford and Mark Crovella. Critical path analysis of tcp transactions. In *SIGCOMM*, 2000.
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. M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.
9. 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.
10. Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abdel-salam 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.

11. 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.
12. Adrian Cockcroft. Watching your web server. The Unix Insider at http://www.unixinsider.com, April 1996.
13. 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.
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. Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of OSDI '96*, October 1996.
18. Fielding, Gettys, Mogul, Frystyk, and Berners-lee. DNS support for load balancing. RFC 2068, April 1997.
19. J.E. Gehrke, S. Muthukrishnan, R. Rajaraman, and A. Shaheen. Scheduling to minimize average stretch online. In *40th Annual symposium on Foundation of Computer Science*, pages 433–422, 1999.
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. Internet Town Hall. The internet traffic archives. Available at `http://town.hall.org/Archives/pub/ITA/`.
23. Mor Harchol-Balter, Nikhil Bansal, Bianca Schroeder, and Mukesh Agrawal. Implementation of SRPT scheduling in web servers. Technical Report CMU-CS-00-170, 2000.
24. Mor Harchol-Balter and Allen Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of SIGMETRICS '96*, pages 13–24, 1996.
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. Gordon Irlam. Unix file size survey - 1993. Available at `http://www.base.com/gordoni/ufs93.html`, September 1994.
27. M. Kaashoek, D. Engler, D. Wallach, and G. Ganger. Server operating systems. In *SIGOPS European Workshop*, September 1996.
28. L. Kleinrock, R.R. Muntz, and J. Hsu. Tight bounds on average response time for time-shared computer systems. In *Proceedings of the IFIP Congress*, volume 1, pages 124–133, 1971.
29. Leonard Kleinrock. *Queueing Systems*, volume II. Computer Applications. John Wiley & Sons, 1976.
30. W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM Sigmetrics*, pages 54–69, 1986.

31. S. Manley and M. Seltzer. Web facts and fantasy. In *Proceedings of the 1997 USITS*, 1997.
32. Evangelos Markatos. Main memory caching of Web documents. In *Proceedings of the Fifth Interntional Conference on the WWW*, 1996.
33. J. Mogul. Operating systems support for busy internet servers. Technical Report WRL-Technical-Note-49, Compaq Western Research Lab, May 1995.
34. V. N. Padmanabhan and J. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28:25–35, December 1995.
35. Vivek S. Pai, Peter Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of USENIX 1999*, June 1999.
36. A.V. Pechinkin, A.D. Solovyev, and S.F. Yashkov. A system with servicing discipline whereby the order of remaining length is serviced first. *Tekhnicheskaya Kibernetika*, 17:51–59, 1979.
37. R. Perera. The variance of delay time in queueing system M/G/1 with optimal strategy SRPT. *Archiv fur Elektronik und Uebertragungstechnik*, 47:110–114, 1993.
38. David L. Peterson and David B. Adams. Fractal patterns in DASD I/O traffic. In *CMG Proceedings*, December 1996.
39. J. Roberts and L. Massoulie. Bandwidth sharing and admission control for elastic traffic. In *ITC Specialist Seminar*, 1998.
40. R. Schassberger. The steady-state appearance of the M/G/1 queue under the discipline of shortest remaining processing time. *Advances in Applied Probability*, 22:456–479, 1990.
41. L.E. Schrage. A proof of the optimality of the shortest processing remaining time discipline. *Operations Research*, 16:678–690, 1968.
42. L.E. Schrage and L.W. Miller. The queue M/G/1 with the shortest processing remaining time discipline. *Operations Research*, 14:670–684, 1966.
43. F. Schreiber. Properties and applications of the optimal queueing strategy SRPT - a survey. *Archiv fur Elektronik und Uebertragungstechnik*, 47:372–378, 1993.
44. Bianca Schroeder and Mor Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. In *9th IEEE Symposium on High Performance Distributed Computing (HPDC '00)*, August 2000.
45. A. Silberschatz and P. Galvin. *Operating System Concepts, 5th Edition.* John Wiley & Sons, 1998.
46. D.R. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26:197–199, 1976.
47. W. Stallings. *Operating Systems, 2nd Edition.* Prentice Hall, 1995.
48. A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.