SWIFT: Scheduling in Web Servers for Fast Response Time

Mayank Rawat and Ajay Kshemkalyani
Univ. of Illinois at Chicago, Chicago, IL 60607
{mrawat,ajayk}@cs.uic.edu

Abstract

This paper addresses the problem of how to service web requests quickly in order to minimize the client response time. Some of the recent work uses the idea of the Shortest Remaining Processing Time scheduling (SRPT) in Web servers in order to give preference to requests for short files. However, by considering only the size of the file for determining the priority of requests, the previous works lack in capturing potentially useful scheduling information contained in the interaction between networks and end systems. To address this, this paper proposes and implements an algorithm, SWIFT, that focuses on both server and network characteristics in conjunction. Our approach prioritizes requests based on the size of the file requested and the distance of the client from the server. The implementation is at the kernel level for a finer-grained control over the packets entering the network. We present the results of the experiments conducted in a WAN environment to test the efficacy of SWIFT. The results show that for large-sized files, SWIFT shows an improvement of 2.5% - 10% over the SRPT scheme for the tested server loads.

1 Introduction

The World Wide Web has probably become one of the most important objects of study and evaluation, and as it is not centrally planned or configured, many basic questions about its nature are wide open. A basic question that has motivated much work is: how do we make the slow Web fast? Typical Web servers today have to service several hundred requests concurrently. Some very popular web-sites like **google.com** receive several million hits per day. In such a scenario, a client accessing a busy Web server can expect a long delay. This delay is due to several factors like queueing delays at Web servers, the current network conditions, and delays introduced by network protocols like TCP due to slow start, congestion, and network loss. All these delays, together, comprise the *response time* which is defined as the duration from when a client makes a request until the entire file is received by the client.

Traditional human factors research shows the need for response times faster than a second. Unfortunately, subsecond response times on the Web are yet to be achieved. Currently, the minimum goal for response times should, therefore, be to get pages to users in no more than 10 seconds, as that is the limit of people's ability to keep their attention focused while waiting [15]. Our focus is on what we can do to improve the client response time, and thereby better the user's experience. In this paper, we propose a new scheduling algorithm on the Web server, called SWIFT. The algorithm keeps with the response time requirements set in [15] and focuses on reducing the delays.

Critical path analysis done by Barford and Crovella [6] for web transactions has shown that for a busy server, delay at the server is the predominant factor (up to 80%) in the overall response time. The experiments monitored transactions that were restricted to three sizes of files: 1KB, 20KB, and 500 KB corresponding to *small*, *medium*, and *large* files, respectively. The authors noted that since small files were common in the web, performance improvements were possible by improving the server's service for small files; this result corroborates the validity of size-based scheduling like [3]. The other observation made in [6] was that for medium-sized files, network delays (network variation and propagation) dominated when server load was low. However, when the server was highly loaded, both the network and the server contributed comparably to the delay. Finally, for large files, the network delays were the dominating component in the overall response time. All these observations suggest that one cannot ignore the effect of network delays while scheduling medium and large-sized files. It is for these reasons that we decided to account for network delays in designing scheduling policies for Web servers.

The proposed idea, SWIFT, is based on the Shortest Remaining Processing Time (SRPT) scheduling policy which has long been known to be optimal for minimizing mean response time [17, 18]. However, the belief that SRPT penalizes large jobs in favor of shorter jobs led to the adoption of 'fair' scheduling policies such as Processor-Sharing (PS), which did little to improve the mean response time. Recently, Mor Harchol-Balter et. al. [3, 5] have taken up work on SRPT scheduling in Web servers and they opened a trend for so-called 'unfair' scheduling policies. Bansal et. al. [5] give the theoretical basis for the viability of an SRPT scheme in the current scenario. The important assumption made is that file size distributions in the Web exhibit **heavy tails**, including files requested by users, files transmitted through the network, and files stored on servers [11, 13].

For our implementation, we consider heavy-tailed web workloads and focus on *static* requests only. The latter assumption is based on the measurement study done by Man-

Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA'03)
0-7695-1938-5/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

ley et. al. [14], where they suggest that most servers process relatively little dynamic (CGI) content, and the traffic generated accounts for a small fraction of the site's traffic.

Unlike previous work which focuses only on the size of the request, our idea is to prioritize the requests based on the size of the request and the propagation delay. The size of the request can be approximated by the size of the file and the propagation delay can be proportional to the average round-trips. The server can estimate the *average round-trip time* (RTT) after receiving the web requests. This is commonly done in today's generation of Web servers. We, however, use a trace-driven web request generator and for experimental purpose, assume that the RTT is embedded in the request. Based on these two parameters, the SWIFT scheduling policy can be implemented. We conducted experiments in a WAN environment and used the Apache Web server (the Feb. 2001 NetCraft Web server survey found that 60% of the web sites on the Internet are using Apache). The clients made use of a trace-based workload to generate requests. The experiments were repeated at different server loads. The server load in our experiments was defined as the load at the bottleneck resource. As in [3], this is the network link out of the Web server [12], on which the client requests arrive. Memory is not a bottleneck because the RAM on the server can be available at low costs. Moreover, servers can use File System Cache to service most of the requests from the cache, e.g., IIS 5.0 uses 50% of the available physical memory as cache. This reduces the number of *hard* page faults. Similarly, processors are not bottleneck resources as their speeds are doubling every 18 months as per Moore's Law, thereby ensuring that the CPU is much faster than network connections. As network bandwidth is the most likely source of a performance bottleneck for a site consisting of static content, we define load as the fraction of bandwidth used on the network link out of the Web server. Our experiments show that for large files, SWIFT shows an improvement of 2.5% - 10% over the SRPT scheme in [3] (which is the most closely related work and along which this paper is structured) for the tested server loads.

2. Related work

There has been considerable research on minimizing the response time of web requests. Some of this focuses on reducing *server delays*. This includes work towards building efficient HTTP servers or improving the operating system of servers. Other research addresses how to reduce the *network latency*. This can be accomplished by caching web requests or improving the HTTP protocol itself. Our work focuses on reducing delays at the server and hence, touches on several related areas in server design, and in the theory and practice of scheduling in operating systems.

In our work, we focus on servers that serve static content, i.e., files whose size can be determined in advance. We do this based on the measurement studies undertaken in [14], which suggested that most servers serve mainly static content. The work also noted that dynamic content was served mainly from a relatively small fraction of the servers in the web. Since Web servers can serve dynamic content as well, a heuristic policy to estimate the size of files can be employed. We can then use these estimated file sizes to improve the performance of shorter requests. In this case, our methods are less directly applicable. However, our technology may be expanded to schedule dynamic pages also.

It is well understood from traditional operating system schedulers that if the task sizes *are* known, the work-conserving scheduling strategy that minimizes mean response time is Shortest Remaining Processing Time first - SRPT. Schrage and Miller first derived the expressions for the response time in an $M/G/1/SRPT$ queue [18]. Besides SRPT, there are many algorithms in the literature, designed for the case where the task size is known.

The earliest paper related to our work is that of Bender et al. [9]. This paper discusses a size-based scheduling on the Web and raises an important point: in choosing a scheduling policy, it is important to consider not only the scheduling policy's performance, but also whether the policy is fair. Roberts and Massoulie [16] consider bandwidth sharing on a link and suggest that SRPT scheduling may be beneficial for heavy-tailed (Pareto) flow sizes.

Almeida et. al [2] investigated approaches to differentiated quality of service by assigning priorities to requests based on the requested documents. They presented priority-based scheduling at both the user and kernel-level. In the user-level approach, a scheduler process is included in the Apache Web server. The scheduler restricts the maximum number of concurrent processes servicing requests of each priority. All the modifications are in the application level and as a result, there is no control over what the operating system does when servicing the requests. In the kernel-level approach, the Linux kernel is modified such that request priorities are mapped into priorities of the HTTP processes handling them. The experiments showed that the high-priority requests benefit by up to 20% and the low-priority requests suffer by up to 200%.

Our paper is most closely related to the work done by Mor Harchol-Balter [3], which deals with SRPT scheduling at Web servers. The implementation is at the kernel level and the idea is to give preference to those requests that are short, or have small remaining processing time. Another implementation of the same idea as [3] was proposed earlier by Crovella et. al. in [10]. However, [10] did connection scheduling at the application level only. Although the kernel-level implementation of [3] is better, it does not consider the distance of clients from the server in prioritizing the requests. This drawback is overcome in our work because we consider the round-trip times (RTT is a measure of the distance of client from the server), as well, when making priority decisions of client requests.

Bansal et. al. [5] give the theoretical basis for considering SRPT scheduling and show that for a large range of heavy-tailed distributions, every request performs better under SRPT scheduling as compared to processor-sharing scheduling.

3. Design and implementation

In [3, 12], the authors conclude: "*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.*" Another reason why the bottleneck resource is the bandwidth on the access link out of the web site is that these links (T3, OC3) cost a fortune per month compared to the CPU's speed/$ ratio [3]. As the network link is the bottleneck resource, the SWIFT scheduling is done on the bandwidth on the access link out of the Web server. We consider only static content as in [3].

Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA'03)
0-7695-1938-5/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

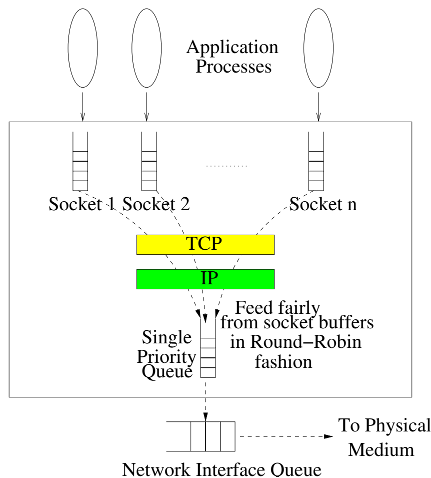

Figure 1. Data flow in Linux [3]

3.1. Differentiated services in Linux

Figure 1 shows the processing of stream sockets (TCP) in standard Linux. These sockets provide reliable two-way sequenced data streams. Linux uses these socket buffers to pass data between the protocol layers and the network device drivers. The packet streams corresponding to each connection drain fairly (in a round-robin fashion) into a single priority queue. Packets leaving this queue drain into a particular device queue. Every network device has its own queueing discipline. The Linux default is FIFO. As we schedule requests based on their size and the RTT of the clients (generating the requests) from the server, we need several priority levels to which the data packets corresponding to the requests can be directed. To do so, Differentiated Services (DiffServ) [1] needs to be integrated into the Linux Kernel. This can be done by building the Linux kernel with the following configuration options enabled [1].

- User/Kernel Netlink Socket (*CONFIG_NETLINK*)
- QoS and/or Fair Queueing (*CONFIG_NET_SCHED*)
- The simplest PRIO Pseudoscheduler (*CONFIG_NET_SCH_PRIO*)

By configuring the kernel for PRIO Pseudoscheduler, the queueing discipline of the network device is changed to priority-based rather than a FIFO default configuration.

Once the DiffServ patch [1] (known to work with Ver. 2.2.14 and above of Linux kernel) is applied, the *iproute2* package is installed. This package includes the Linux Traffic Control utility – *tc* which is used to switch from a single priority queue to 16 priority queues. Figure 2 shows the data flow with the DiffServ patch, as in [3]. There are now 16 priority queues, numbered 0 through 15. The lowest numbered priority queue (number 0) has the highest priority; priority queue 15 has the lowest priority. The new SWIFT module is used to determine the priority of the connection based on the size of the request and round-trip time parameters. Connections of priority i go to the ith priority queue. Packets are drained from the multiple priority queues into the network device queue in a *prioritized* fashion. priority queue i is allowed to flow only if queues 0 to $i-1$ are empty.

Some additional fixes were recommended in [3]. We use these fixes in our scheme. The startup time for connections is an important component of the total response time.

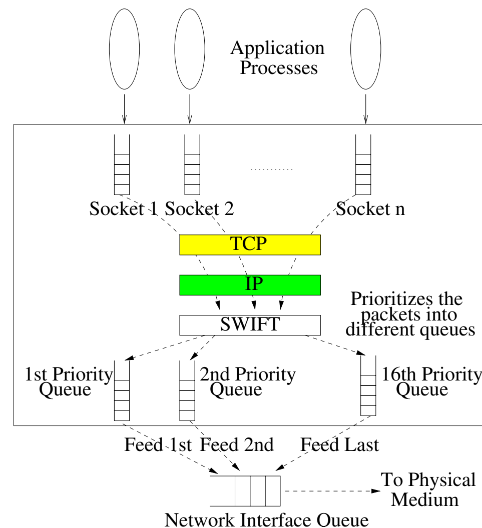

Figure 2. Data flow in Linux after changes [3]. SWIFT module is added new in this figure.

This is more true for shorter requests. In SRPT scheduling, during the connection startup, it is unclear whether the request would be large or small. “The packets sent during the connection startup might therefore end up waiting in long queues, making connection startup very expensive” [3]. Hence, the SYNACKs must be given the highest priority queue 0. Also, it must be ensured that when assigning priority queues, no sockets are assigned to priority queue 0, but are assigned to queues of lower priority than queue 0. As the SYNACKs make up only a negligible fraction of the total load, giving them high priority does not degrade the performance, and also reduces their loss probability.

3.2. SWIFT: Algorithm and implementation

A key characteristic of DiffServ is that flows are aggregated in the network, so that core routers only need to distinguish a comparably small number of aggregated flows, even if those flows contain thousands or millions of individual flows. Once the Linux kernel is compiled with the DiffServ support, we made changes to the Web server code for prioritizing connections. As mentioned earlier, the RTT which is a measure of propagation delay is assumed to be embedded in the request. The web-server, on receiving an HTTP GET, determines the priority level of the request based on its size and the embedded RTT. We give weights to the size of the request and propagation delay parameters, based on observations of their magnitude in [6]. We then use the weighted sum of these two parameters to determine the priority level of the request. Apache sets the priority of the socket by calling *setsockopt*. After every IOBUFSIZE (8192) bytes of the file are sent, a check is made whether the priority level of the request has fallen below the threshold for its current priority level. If it is so, the socket priority is updated with another call to *setsockopt*.

Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA'03)
0-7695-1938-5/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

3.2.1 Determining weight criteria

Barford and Crovella [6] have done detailed work in pinpointing where delays are introduced into applications like HTTP by using *critical path* analysis. We use their observations to heuristically determine weights for the size of the request and propagation delay.

- *Small* files (0 - 20KB): During low load conditions, the network delays dominate. When the server load is high, the server delays can be responsible for as much as 80% of the overall response time. We therefore choose a weight of 0.8 for request size and 0.2 for propagation delay.
- *Medium-sized* files (20 - 500KB): Network delays dominate when the server load is low; but when server load is high, the contribution to delay from network and server are comparable. So for these files, we assign a weight of 0.6 and 0.4 to request size and propagation delay, respectively.
- *Large* files (500KB and above): Network delays dominate transfer time, irrespective of the load at the server. We use the weight value of 0.75 for propagation delay and 0.25 for the request size.

3.2.2 Cutoffs for RTT and file size

The requests are partitioned into priority classes based on the rules given in [3]. Unlike [3] which has a single priority array based on the file size to store the size cutoffs, we maintain two priority arrays – one corresponding to the file size (request size) and the other corresponding to the RTT (propagation delay). Let us first discuss the cutoffs for file size, denoted as $s_1 < s_2 < ... < s_n$. Under the assumption that workload is heavy-tailed, we have [3]:

- The lowest size cutoff s_1 is such that 50% of requests have size smaller than s_1.
- The highest size cutoff s_n is such that only the largest 2% of the requests have size greater than s_n.
- The in-between cutoffs have logarithmic spacing between them.

The cutoffs for propagation delay (the new parameter we introduce), denoted as $r_1 < r_2 < ... < r_n$, are much easier to determine. We used the average RTT from a location as an estimate of the propagation delay incurred during transfer of data from the server to that location. To date, no work has been done which shows the distribution pattern of propagation delay for web traffic and so we simply assign higher priorities to larger RTTs. For our experiments we have used 16 priority levels. The lowest size cutoff r_1 is for client locations that are less than 10ms from the server. The highest cutoff r_n is chosen to be 350ms. The intermediate cutoffs are at equal intervals. Higher cutoffs are given higher priority than lower cutoffs. Thus, locations like Urbana (RTT 5.6ms), which are close to Chicago, are given lesser priority. while Berkeley (RTT 50ms), is given higher priority. Likewise, locations in U.K (RTT 115ms) are rated more important. In future, studies on propagation delay characteristics can be used to attain cutoffs that are more representative of the web traffic.

3.2.3 Algorithm

The SWIFT algorithm works as follows.

1. During connection set-up phase, a request is given a socket with the highest priority (priority 0). (Section 3.1.)
2. The request size and RTT of the client-generated request are then determined.
3. Corresponding to the size of the file, the weights for the request size and propagation delay are looked up.
4. The weighted sum of these two parameters determines the priority of the socket corresponding to the request.
5. After every IOBUFSIZE (8192) bytes of the file are transmitted, a running count of the bytes sent is saved. A call is then made to determine the new weights and the priority of the socket is changed *dynamically*.

The *setsockopt* system call is used to assign priority to the socket corresponding to the request. As only a limited number (equal to the number of priority queues) of such calls are made, they do not contribute much overhead. The steps require determining the priority level of the request based on its size and the embedded RTT in the request.

To determine the algorithm complexity, we analyze steps 2 to 4. Step 2 determines the size of the file requested by the client and the RTT of the client generating the request. For static requests, the size of the file can be determined trivially when the GET action fired by the client is received by the server. As our trace-driven experiments made a simplistic assumption that RTT was embedded in the request, it took constant time to determine the RTT. In a more realistic scenario, the server measures the RTT *on-the-fly*. A way to do this efficiently is to use the TCP timestamp option (TSopt) - RFC 1323. There are two 4-byte timestamp fields. *Timestamp Value* (TSval) contains the current value of the timestamp clock of the TCP sending the option and *Timestamp Echo Reply* (TSecr) is the value of the most recent timestamp option that was received when the ACK bit was set. Thus, by sending the TSopt in an initial SYN segment the server can calculate the RTT with a single subtraction. Steps 3 and 4 look up the RTT and requested file's size in the static array of cut-offs (Section 3.2.2). This has a worst-case complexity of O($log_2 n$), where n is the size of the array or the number of priority levels. Once the individual priority levels corresponding to RTT and request-size cutoffs are determined, the algorithm makes use of weights (Section 3.2.1) to compute the weighted sum that determines the priority of the socket corresponding to the request.

The overhead is bounded by $\Theta(log_2 n)$ + $\Theta(1)$, as $\Theta(1)$ is the time taken for the arithmatic to calculate RTT and weighted sum. As the bottleneck is the look-up on the array which is small ($n \leq 16$), the solution takes time $\Theta(1)$.

Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA'03)
0-7695-1938-5/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

4. Experimental Setup

4.1. Testbed description

We used the Linux 2.4.18 operating system on a Dell Intel Pentium 3 machine which ran the Apache Web Server 1.3.14. The main memory of the server was large enough to fit all the files requested by the clients, during the experiments. There were two more PCs (Dell Intel Pentium 3) that were used to generate local load using the Scalable URL Reference Generator (SURGE) [8]. We chose SURGE because it generates analytically-based Web reference streams that match a realistic Web workload. The HTTP requests generated to the server are such that they follow empirically measured properties of web workloads like request size distribution, server file size distribution, relative file popularity, etc. These additional two machines that were used solely to generate background traffic were connected by a 10 Mbps switched Ethernet to the server.

The client systems were located at two off-site locations, in Austin (RTT: 30ms) and Seattle (RTT: 50ms). These client machines generated the monitored requests to the server. The Client 1 at Austin (running Debian 2.2) was 15 hops from the server in Chicago. The Client 2 at Seattle (running Red Hat Linux 7.1) was 16 hops away. The path to both the client locations included UCAID/Internet 2 Abilene and StarTap research networks. The clients used a modified version of SCLIENT [4] to make requests. SCLIENT is a web request generator that provides a scalable means of generating realistic HTTP requests, including peak loads that exceed the capacity of the server. In the modified version of SCLIENT, the clients made use of a trace-based workload to generate requests. The *sclient* tool read in traces and made requests to the server according to the arrival time and file names given in the trace. In order to simulate the effects of the varying client locations from a single location *e.g.*, Austin (RTT: 30ms), we needed to introduce artificial delays. So for example, to simulate the effects of a location with an RTT of 50ms, we needed to introduce additional delays amounting to 20ms at our client location in Austin. This would then give the effect that the request from Austin was generated from further away.

4.2. Workload description

As our experiments were trace-driven, we downloaded our trace-based workload from the IETF traffic archive. We made use of the same workload as used in [3]. This is an actual trace of the Soccer Worldcup 1998 collected between April 30,1998 and July 26, 1998; and consists of 1.3 billion Web requests made to 33 different World Cup HTTP servers at four different geographic locations. Our workload consisted of a part of a single day trace. Most of the one-day trace we used contained static requests. Each entry of the trace described a request made to the server and included the time of the request, client identifier, URL requested, the size of the request, the GET method contained in the client's request, HTTP version and other status information, the type of the file requested (*e.g.,* HTML, IMAGE, etc.), and the server that handled the request.

For the experiments, we used the busy hour (10 – 11a.m) of the trace. This hour consisted of about 1 million requests, during which close to a thousand files were requested. The files requested ranged from 40 bytes to 2 MB and they followed the heavy-tailed distribution. The main memory on the server was large enough to cache all the files in the file set so after the initial request (disk reads), all the subsequent requests for a file were served from the main memory. Thus, the disk was not a bottleneck resource.

To study the effects of server load, we used the local load generators (SURGE) on the server side of the testbed to generate either light load (40 SURGE equivalents) or heavy load (500 SURGE equivalents). Each SURGE equivalent is a single bursty process in an endless loop that alternates between making requests for Web files, and lying idle [8].

5. Results

Figures 3, 4, 5, and 6 show the mean response time as a function of the percentile of the request size distribution. This set of graphs is for the client in Austin. For the client in Seattle we have Figures 7, 8, 9, and 10. For each client location, we have four graphs corresponding to a load of 0.5, 0.7, 0.8, and 0.9. Each graph shows a set of lines for an average RTT of 50ms, 100ms, 150ms, and 250ms. These correspond to locations of client that range from near to a very large distance from the Web Server in Chicago. As explained in Section 4, all the client requests are in fact generated in Austin and Seattle.

Every request generated to the server from the client sites was repeated five times and the response time given in the graphs was calculated as an average of these five runs.

The graphs for clients locations at Austin and Seattle are similar in nature but the response time for a given percentile of job size and RTT under a given load may be different in some cases. This is because the measurements are dependent on the queuing delays in the intermediate routers between client locations and the server in Chicago. Also, the bandwidth of the network link out of the client locations can affect the response time. However, this should not affect the validity of our conclusions as our goal was to compare SWIFT with existing algorithms, hence we are more concerned with the percentage differences.

The graph shows two curves, one for SWIFT, one for SRPT. For lower-sized file requests, the SWIFT scheme is similar to the SRPT; response times are not adversely affected by much. However, for large-sized requests, there is 2.5% - 10% improvement with SWIFT vis-a-vis SRPT. We make the following observations from the graphs.

- Nearly, 40% to 80% of the jobs (the larger ones) benefit with the SWIFT scheme compared to the SRPT scheme, whereas for the others (the smaller ones), the response times are not adversely affected much.

- As the distance of the client increases from the server, a larger percentage of the jobs shows improvement for server loads of 0.5, 0.7 and 0.8.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

- Under server load of 0.5 and 0.7, the percentage improvement for large files increases with increasing RTT.

By decreasing the response time for larger files while retaining the general features of SRPT, the user's experience can be bettered. Distant requests for large files also begin to receive a response earlier.

6. Conclusions and discussion

The size-based SRPT scheduling presented by Mor Harchol-Balter et. al. [3] is, to date, the best known method to minimize the client response time. By using an additional parameter, the propagation delay, to prioritize requests, initial experiments show a reduction in the response time for large files. We see an improvement in response time for all job sizes greater than the bottom 40 percentile under low load. We also see improvement for all job sizes greater than the bottom 20 percentile under high load. SWIFT performs better than SRPT as the distance of the client increases from the server. The percentage improvement of SWIFT over SRPT ranges from 2.5% to 10% for larger-sized job requests, with the percentage improvement being lower under high load conditions than under low load conditions. These gains do not adversely impact by much the response time for small-sized job requests. This is surprising and the algorithm needs to be analysed carefully to explain these experimental results.

Concerning unfairness to large requests under heavy load, SWIFT still shows improvement compared to SRPT. But when we compare SWIFT and SRPT to standard Linux with an unchanged web server, the mean response time for the largest size request goes up. But as these requests are so large, their response time is not affected much.

SWIFT leveraged the useful scheduling information contained in the interaction between networks and end systems, by making the scheduling decisions based on both the distance of the client from the server and the size of the request. During the course of the implementation of the algorithm, we examined several weight criteria for propagation delay and request size. Our choice of criteria for assigning weights is based on the *critical path* analysis of HTTP application done by Barford et. al. in [6]. We do not claim that this choice of weights is optimal, but after experimenting with different weights, we found that this choice gave good results. The assignment of weights needs formal investigation. Our approach can also benefit from the studies on the distribution of propagation delay patterns in web traffic. This requires further study.

Another goal is to experiment with alternative means of prioritizing requests, such as using the knowledge of client's bandwidth to determine priorities for requests, and using some algorithms from the area of 'locality aware' request distribution policies.

SWIFT is currently limited to scheduling static requests. However, an increasing number of Web servers today serve dynamic content as well. In such a scenario, a heuristic policy to improve the performance of short tasks needs to be employed. Future work can enhance our technology to also schedule dynamic pages.

References

[1] Differentiated services in Linux, *http://diffserv.sourceforge.net*

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

[3] M. Harchol-Balter, N. Bansal, B. Schroeder, and M. Agrawal, Size-based scheduling to improve web performance, *To appear in ACM TOCS*. Currently available as CMU Technical Report: CMU-CS-00-170, "Implementation of SRPT Scheduling in Web Servers".

[4] G. Banga and P. Druschel, Measuring the capacity of a Web server, *Proc. of the USENIX Symposium on Internet Technologies and Systems*, pp. 61-71, Monterey, Dec. 1997.

[5] N. Bansal and M. Harchol-Balter, Analysis of SRPT scheduling: Investigating unfairness, *Proc. SIGMETRICS 2001*, pp. 279-290, June 2001.

[6] P. Barford and M. E. Crovella, Critical path analysis of TCP transactions, *Proc. 2000 ACM SIGCOMM Conference*, Stockholm, pp. 127-138, Sept. 2000.

[7] P. Barford and M. E. Crovella, Measuring web performance in the wide area, *ACM Performance Evaluation Review*, vol. 27, no. 2, pp. 37-48, August 1999.

[8] P. Barford and M. E. Crovella, Generating representative workloads for network and server performance evaluation, *Proc. ACM SIGMETRICS 1998*, pp. 151-160, June '98.

[9] M. Bender, S. Chakrabarti, and S. Muthukrishnan, Flow and stretch metrics for scheduling continuous job stream, *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 270-279, 1998.

[10] M. E. Crovella, B. Frangioso, and M. Harchol-Balter, Connection scheduling in web servers, *USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, pp. 243-254, Oct. 1999.

[11] M. E. Crovella, M. S. Taqqu, and A. Bestavros, Heavy-tailed probability distributions in the World Wide Web, *Chapter 1 of A Practical Guide To Heavy Tails*, pp. 3-25, Chapman & Hall, New York, 1998.

[12] B. Curry, G. V. Reilly, and H. Kaldestad, The art and science of web-server tuning with internet informations services 5.0, *http://www.microsoft.com/technet/treeview/default.asp ?url=/technet/prodtechnol/iis/maintain/optimize/iis5tune.asp*, 2001.

[13] A. B. Downey, Evidence for long-tailed distributions in the Internet, *ACM SIGCOMM Internet Measurement Workshop*, Nov 2001.

[14] S. Manley and M. Seltzer, Web facts and fantasy, *USENIX Symp. on Internet Technologies and Systems (USITS)*, 1997.

[15] J. Nielsen, Need for Speed, *http://www.useit.com/alertbox/9703a.html*, Mar 1, 1997.

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

[17] L. E. Schrage, A proof of the optimality of the shortest remaining processing time discipline, *Operations Research*, 16:678-690, 1968.

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

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

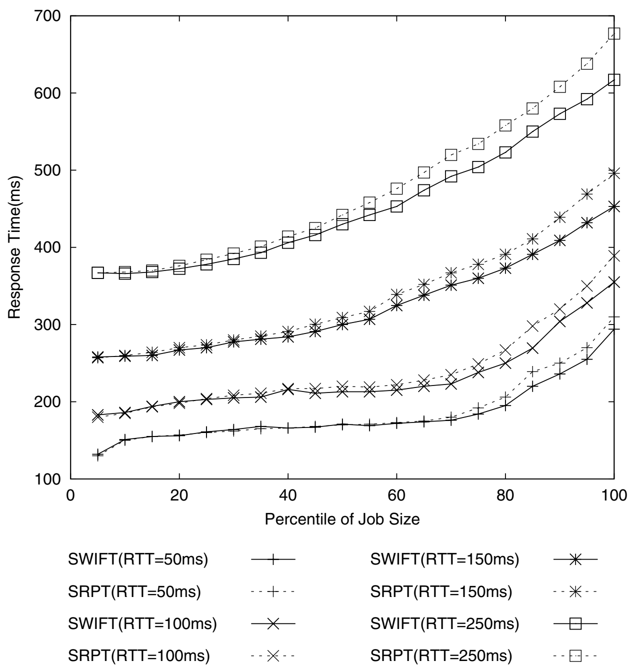

Figure 3. SWIFT Vs SRPT for server load 0.5 [client at Austin]

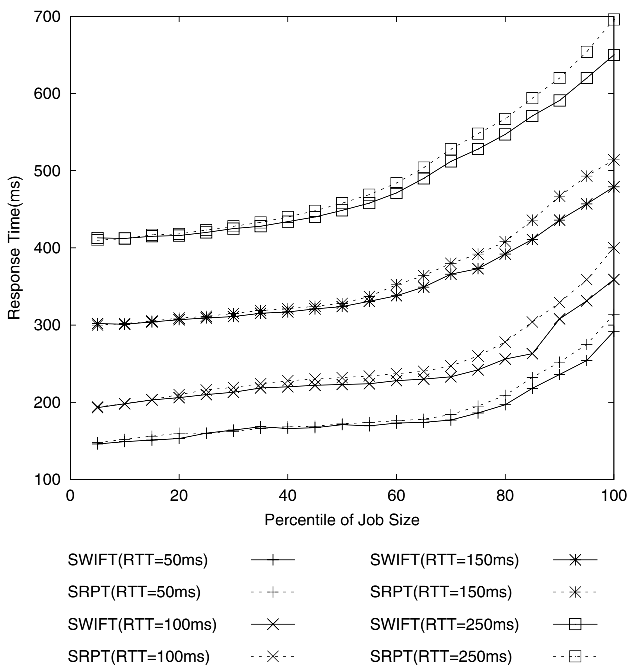

Figure 4. SWIFT Vs SRPT for server load 0.7 [client at Austin]

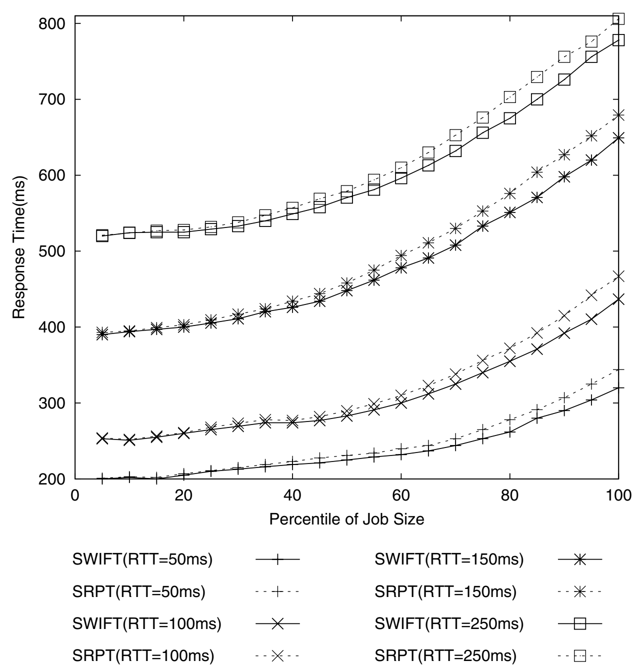

Figure 5. SWIFT Vs SRPT for server load 0.8 [client at Austin]

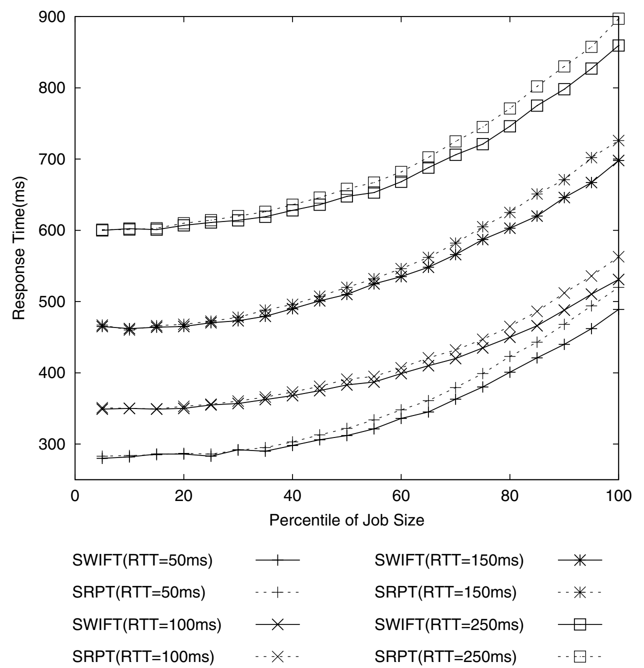

Figure 6. SWIFT Vs SRPT for server load 0.9 [client at Austin]

0-7695-1938-5/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.

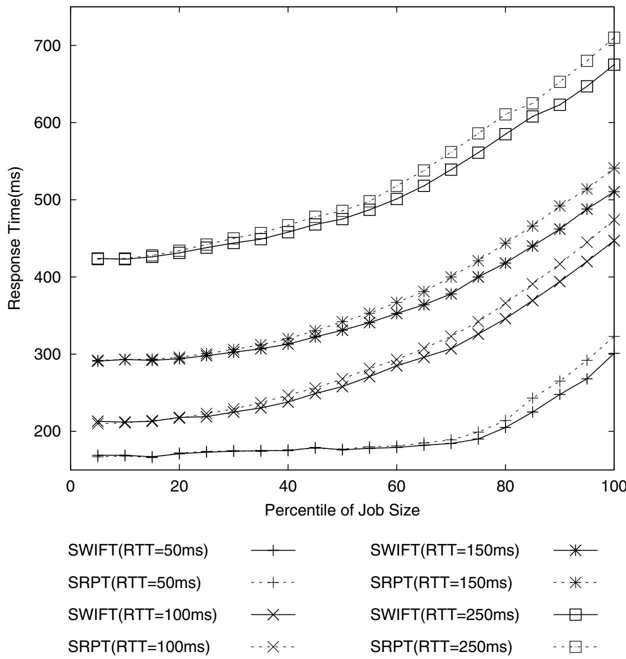

Figure 7. SWIFT Vs SRPT for server load 0.5 [client at Seattle]

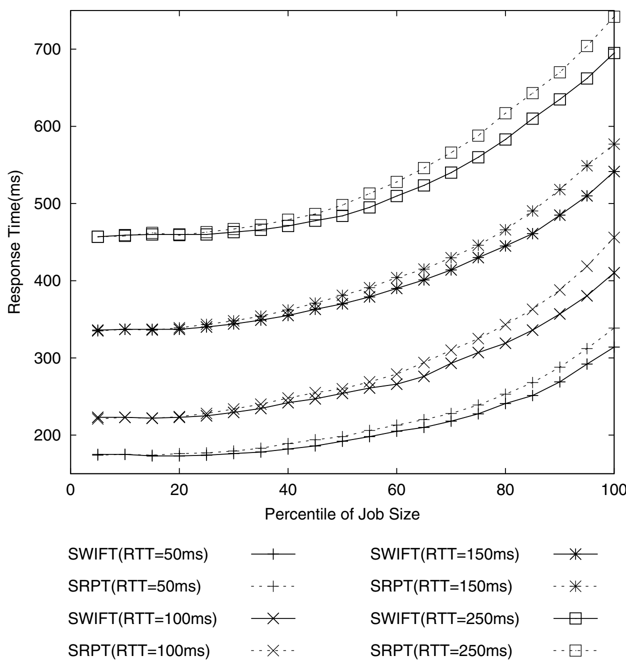

Figure 8. SWIFT Vs SRPT for server load 0.7 [client at Seattle]

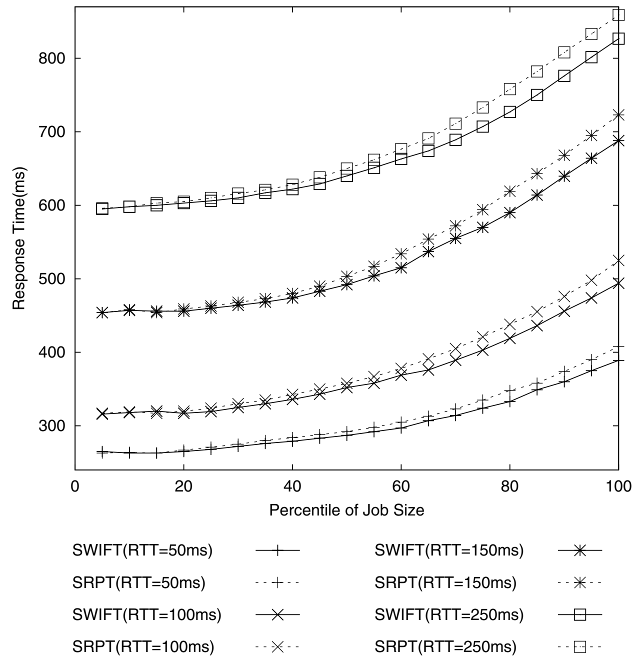

Figure 9. SWIFT Vs SRPT for server load 0.8 [client at Seattle]

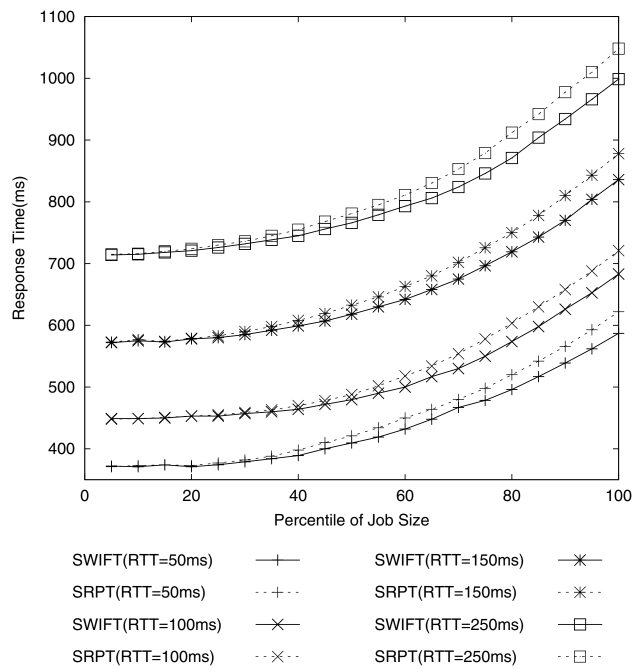

Figure 10. SWIFT Vs SRPT for server load 0.9 [client at Seattle]

Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA'03)
0-7695-1938-5/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Illinois at Chicago Library. Downloaded on June 22,2020 at 22:39:47 UTC from IEEE Xplore. Restrictions apply.