Operating Systems Support for the Evolution of Software: An Evaluation Using WWW Server Software

Sukanya Suranauwarat Hideo Taniguchi
Graduate School of Information Science and Electrical Engineering,
Kyushu University, Fukuoka 812-8581, Japan
{sukanya,tani}@csce.kyushu-u.ac.jp

Abstract

We believe that improving an operating system's support for the evolution of software is vital to our goal of reducing the significant sum spent on adapting existing software to changing user requirements, especially to improve the performance of software. Therefore, we proposed the idea that by increasing an operating system's abilities to observe the software's execution behavior and evolve its execution behavior using observed results, an operating system could adapt existing software to changing user requirements without making any changes to the software. We integrated the above abilities into a CPU scheduling mechanism in an operating system, and verified the usefulness of our idea using existing software, i.e., a World Wide Web (WWW) server. In this case, our scheduling mechanism alters the execution behavior of a WWW server by giving preferential use of the CPU resource to server processes handling HTML file requests. This allows the user requirement, which is the enhancement of response time during periods of high demand, to be satisfied. In order to determine which processes are server processes handling HTML file requests, we introduced scheduling parameters SLP and RW. In this paper, we describe how we predicted and updated parameter RW based on the observed execution behavior of a WWW server, and present the experimental validation of our method.

1. Introduction

Estimates of the annual cost of maintaining software already written vary widely, but range between 50% and 80% of the total software development budget in any given period [1]-[3]. In other words, for every dollar spent on developing software, it is claimed that at least a further dollar will be spent on software maintenance during the life of the software. Therefore, software maintenance is an area where even small improvements to the process can potentially reap significant benefits [4].

When looking for a way to reduce the maintenance cost, it is worth while to take a look at the cost spent on each maintenance activity; and studies have shown that over 50% of the cost is spent on changes to accommodate changing user requirements. Changes in user requirements are inevitable. Software models part of reality, and reality changes. So the software has to change too. It has to evolve. Keep this in mind, our goal is to reduce the significant sum spent on the evolution of software due to changing user requirements, especially to improve the performance of software. And this goal could be achieved by using the following idea. Our idea is that by increasing an operating system's abilities to *observe* the software's execution behavior and *evolve* its execution behavior using observed results, an operating system could *adapt* existing software to changing user requirements *without making any changes* to the software. In other words, by using these abilities, an operating system could optimize a software's execution behavior allowing user requirements to be satisfied without any changes to the existing software.

We decided to integrate the above abilities into a CPU scheduling mechanism in an operating system, and verify the usefulness of our idea using existing software, i.e., a World Wide Web (WWW) server. In this case, our scheduling mechanism alters the execution behavior of a WWW server by giving preferential use of the CPU resource to any process of a WWW server that is predicted to be a server process handling an HTML file request. This allows users to view text and the general layout of a WWW page in a timely manner during periods of high demand resulting in the user requirement, the enhancement of response time when there is a heavy demand on the server, to be satisfied without making any changes to the existing server software.

In order to predict or determine which processes are server processes handling HTML file requests, we introduced scheduling parameters SLP and RW. A description of how to predict and update SLP based on the observed execution behavior of a WWW server that includes a performance evaluation has already been reported in [5]-[7]. And, our experimental results show that the response time of the server is improved when SLP is set using the proposed method. However, the degree of improvement varies according to how RW is fixed. To be more precise, the more the value of RW reflects the actual behavior of server processes handling HTML files, the better the performance enhancement will be. Since the execution behavior of a WWW server generally changes according to the service demand placed on it (i.e., the files it is requested), fixing RW at some values will only slightly improve the performance. Therefore, in order to always obtain the best performance, RW should also be predicted and updated automatically to the changing server's execution behavior that we observe.

In this paper, we describe how we predicted and updated parameter RW and present experimental validation of our method.

The remainder of this paper is organized as follows. Section 2 provides background information on the WWW. Section 3 is a brief overview of our scheduling mechanism. Section 4 describes how to predict and update RW. Section 5 describes our experiments and explains the results we obtained. Section 6 offers our conclusions and future work.

2. The WWW

The WWW is based on the client-server model [8],[9]. That is, users access WWW pages provided by WWW servers via WWW clients, mainly browsers. WWW pages are written in the HyperText Markup Language (HTML) and stored on a disk as text files called HTML files. An HTML file contains text data that users will view and HTML tags that specify structure for the text data as well as formatting hints. Because an HTML file uses a text representation, non-text data such as images are not included directly in the HTML file. Instead, a tag is placed in the HTML file to specify the place at which an image should be inserted and the source of the file that stores it (Image file). HTML and Image files account for more than 90% of the total requests to a server [10],[11]. Therefore, the WWW page in this paper consists of an HTML file and an Image file.

WWW clients. A user can access the information on the WWW by using a browser, such as Netscape Navigator, Mosaic, or lynx. When the user selects a WWW page to retrieve (usually, by clicking a mouse on a hyperlink), the browser creates a request to be sent to the corresponding WWW server and then waits for a response. When the response arrives, the browser interprets and processes the HTML file sent back by the server, and then displays the text data for the user to view. During the interpretation, if the WWW page also contains other types of data such as an image, then the browser will request the Image file from the WWW server.

WWW servers. The purpose of a WWW server is to provide WWW pages to WWW clients that request them. The server software we used is *Apache* version 1.2.5, the pre-forking model server. In this model, a *master* process pre-fork a pool of *child* server processes to handle requests. However, the master process does not handle any part of the request. In this paper, we refer to each child server process as a server process.

3. Overview

Our scheduling mechanism is composed of *two* parts: the *logging mechanism* and the *process control mechanism*. We observe the execution behavior of a software application and create/update the observed results called *PFS (Program Flow Sequence)* through the logging mechanism. And we alter the software application's execution behavior using this PFS through the process control mechanism. The following sections describe each mechanism in more detail, with emphasis on our example software application, which is a WWW server.

3.1. Logging Mechanism

When a WWW server is running, a log is collected. A log is a sequence of entries describing process identifier, process state and time. And based on this log, a sequence called PFS, which is the predicted behavior of a process, is created or updated for each server process. PFS is a sequence of entries describing process state and time spent.

3.2. Process Control Mechanism

As the demand placed on a WWW server grows, the number of simultaneous requests it must handle increases. As a result, users see slower response times. To be more precise, it takes a longer time for the text data stored in an HTML file to show up on browsers so that users can view the contents. This situation could be one in which it is most desirable for users to improve the response time of a WWW server, since they tend to get frustrated if it takes a long time to view a WWW page [12]. Hence, in such a situation, our scheduling goal is to display the text data for the user to view as soon as possible while the images are trickling in and also to allow the user to stop loading if the page is not sufficiently interesting to warrant waiting. A method to achieve this goal is described below.

Most processes, except the currently executing process (i.e., process that is in the *run* state), are in one of two queues: a ready queue or a sleep queue. Processes that are waiting for the CPU to become available (i.e., in the *ready* state) are placed on a ready queue, whereas processes that are blocked awaiting an event (i.e., in the *wait* state) are located on a sleep queue associated with the event. When a process is blocked awaiting an event to happen, if the resources (e.g., a hard disk) needed for the event are being used by any other process, then that process needs to wait for those resources to become available. Next, that process needs to wait for the operation (e.g., input/output) it initiated to be completed. By reducing the time waiting for the CPU to become available in the ready queue or for the resource needed for an event to become available in the sleep queue, we can achieve an enhanced response time. According to this, we proposed the scheduling policy that when a CPU (a hard disk or a network communication) becomes bottlenecked, any server process handling an HTML file will be moved to the head of the ready queue (sleep queue associated with the event). Note that the bottleneck of the CPU mentioned in this paper is the situation in which the CPU is busy and there is more than one process waiting in the ready queue.

When we discussed the process control mechanism that implements the above policy focused on the bottleneck of the CPU, we had *two* problems: *how to detect which processes are server processes handling HTML files*, and *how to operate the ready queue*. To answer these questions we found that we needed to look at the detailed execution behavior of a WWW server. We analyzed the behavior of server processes based on PFS and found that a server process handling an HTML file is the process that waits for a request from a browser, and the time it waits for a request is relatively long. Also, after waiting for a request, it tends to change between run state and wait state a number of times. This number of changes is proportional to the size of files it handles, i.e., HTML files (which are generally smaller than Image files). In other words, any server process handling an HTML file has two characteristics: After waiting for a long time in the wait state (characteristic 1), it tends to cycle between run state and wait state a number of times but fewer times than that of a server process handling an Image file (characteristic 2).

To deal with the first problem, we introduced two parameters into our mechanism in order to determine which processes are server processes handling HTML files: long wait threshold (its value is denoted by SLP) and run state/wait state threshold (its value is denoted by RW). If the time spent by a process in the wait state before moving to the run state is more than SLP, and the number of times the process changes between run state and wait state is less than RW, then we determine that it is a server process handling an HTML file. By these two parameters, we can detect which process appears to be a server process handling an HTML file.

To deal with the second problem, our mechanism puts any process that has characteristic 1 at the head of ready queue and moves that process to the back of the ready queue when that process loses characteristic 2. The reason processes that lose characteristic 2 are moved to the back of the ready queue is that in the case that server processes handling Image files are mistaken as server processes handling HTML files because they exhibit characteristic 1. In other words, a server process handling an Image file will be treated as a server process handling an HTML file when it runs after a long wait, until the number of changes between run state and wait state is more than RW.

4. Method of Predicting RW

We analyzed the execution behavior of a WWW server based on its PFS and found that the number of times a server process changes between run state and wait state is proportional to the size of the file it handles (i.e., HTML file or Image file). In other words, the number of changes is proportional to the number of disk accesses required to retrieve the requested files. Since HTML files are generally smaller than Image files, the number of times a server process handling an HTML file changes between run state and wait state is smaller than that of a server process handling an Image file. Therefore, the smallest number of times each server process changes between run state and wait state is determined from its PFS for each period, then RW for the next time period is set to the greatest of these values. We describe how to predict RW, in a more general way, in the following steps:

step 1: When a server process is created, that process will be registered. For each registered process, a PFS is generated.

step 2: The PFS for registered server processes is generated once every period, where the period length is predetermined. The periods are the same for PFS and RW.

step 3: At the end of each period, the PFS for each registered process will be updated. Based on the PFS for each process, we find the smallest number of times each process changed between run state and wait state. The RW for the next time period is set to the greatest of these values.

step 4: When a registered process terminates, it becomes unregistered, in other words, the PFS for that process will no longer be updated.

In step 3, we find the smallest number of times each process changes between run state and wait state, this is actually quite an involved process and requires a more detailed description. First, for each process, each number of times it changes between run state and wait state until the time in the wait state exceeds *SLPmin* (describe later) is stored in an array called *RWbuffer*. Second, we set what the minimum length of time is for a long wait, SLPmin. Third, we set a minimum threshold for RW, *RWmin*, in order to avoid considering the number of changes due to anything but the number of disk accesses for requested files, for example, the number of changes due to multiple server processes calling `accept` system call on the same listening descriptor[1], which requires one or two changes per occurrence. At the end of each time period, from the updated PFS, the number of changes between run state and wait state (*counter*) are counted until the time in the wait state exceeds SLPmin. When SLPmin is exceeded, the counter is compared to RWmin, if it is greater than RWmin then the value is stored in the RWbuffer for that process, otherwise it is discarded. The counter is then reset and it starts counting from where it left off[2]. The minimum values from each process' RWbuffer are compared and RW for the next time period is assigned the largest value. The reasons for using SLPmin, RWmin and RWbuffer are described below. SLPmin is used so that RW can be determined independently from SLP. RWmin is used so that unrelated changes between run state and wait state will be disregarded. RWbuffer is used so that RW for the next time period is based more on a recent history of the processes rather than just the latest period's behavior. Also by using RWbuffer, if processes run abnormally for a short period of time, then RW will not be effected also it tends to cause RW to reflect the behavior of the server processes more accurately.

[1]When the first client connection arrives, all those multiple server processes are awakened. However, only the first process of those processes to run will obtain the connection and the rest will all go back to sleep. Note that if a *lock* of some form around the call to `accept`, so that only one process at a time is blocked in the call to `accept`, then the remaining processes will be blocked trying to obtain the lock.

[2]The counter is independent of the PFS update period (i.e., the counter continues counting even when the time period ends).

5. Experimental Evaluation

We performed experiments to evaluate the effectiveness of our scheduling mechanism when RW is predicted and updated based on PFS automatically every 500 milliseconds. Our mechanism was implemented as modifications to the BSD/OS 2.1 kernel.

5.1. Experimental Setup

In all experiments, the server machine was a 233MHz AMD-K6 PC, with 64MB of memory, running our modified version of BSD/OS 2.1. The client machines were 200MHz Pentium Pro PCs, with 64MB of memory, running BSD/OS 2.1. Our server and client software were Apache 1.2.5. and Netscape Navigator 3.04 respectively. All experiments were conducted in single user mode, and the operating system's I/O buffer cache in the server machine as well as each browser's cache were disabled during the experiments, in order to see the effect of our scheduling mechanism clearly. Also, RWbuffer is set to store up to 5 values. RWmin is set to 2. SLPmin is set to 200 milliseconds.

5.2. Experimental Results

5.2.1. Validation of Our Scheduling Mechanism

In this section, our experiment was designed to verify that our scheduling mechanism with the new embedded function to predict RW works as expected. This experiment was conducted using one client machine and the server machine in a 10 Mbps Ethernet environment. In order to see the effect of our method clearly, we ran a computation intensive background process that bottlenecked the CPU throughout the experiment. And, we set the browser to access the WWW server in such a way that SLP would be predicted 100% correctly, that is, it accessed the WWW server every 30 seconds while SLP was fixed at 20 seconds.

First experiment. In this experiment, we varied the size of the HTML files from 1, 3, 5, and 7 times the original size (1,772 bytes) while the size of the Image file was fixed at 43,770 bytes. For each size of the HTML files, we measured the 5 trial times

Table 1. The number of disk access(es) required to retrieve an HTML file.

size of HTML file	the number of disk accesses	
	consecutive	non-consecutive
1 time (1,772 bytes)	1 time	1 time
3 times (5,316 bytes)	1 time	2 times
5 times (8,860 bytes)	2 times	3 times
7 times (12,404 bytes)	2 times	4 times

of *time1* and *time2*, when RW was fixed at 1, 3, 5 (*RW=1,3,5*) and when RW was set automatically based on PFS (*RW=auto*). Time1 is the time from requesting a WWW page until text data starts displaying. Time2 is the time from requesting a WWW page until image data displays completely. We will refer to the averages of 5 trial times of time1 and time2 as response times of text data and image data respectively.

Figure 1 shows experimental results plotted with the response time on the y-axis normalized by the response time when using a conventional time-sharing mechanism. To analyze the obtained results, the *file read-ahead* operation in the operating system needs to be taken into consideration. Since it will have an effect on the number of disk accesses required to retrieve HTML or Image files, which consequently affects the number of times a corresponding process changes between run state and wait state.

In UNIX systems, files are stored on the disk as a number of blocks, each of which has a size of 4 KB. In order to improve performance, operating systems usually perform a file read-ahead, that is, operating systems will asynchronously prefetch the next block of file data with each read request if a requested file is stored in *consecutive blocks* (in this case, 8 KB of data will be read instead of 4KB). Table 1 summarizes the number of disk accesses required to retrieve HTML files when they are stored in consecutive blocks and when they are not.

Table 1 shows that the number of disk accesses is at most 3 when the size of the HTML files is in the range of 1 to 5 times the original size. As a result, the response times of text data when RW=3 and when RW=5 are about the same as shown in Figure 1(a). In the same way, the response times when RW=auto are improved. However, when the size of the HTML file is 7 times the original size, the improvement when RW=5 is better than when RW=3. This could be because the HTML file in this case is split up into non-consecutive blocks causing the disk to be accessed 4 times (which is more than the RW value of 3) as shown in Table 1.

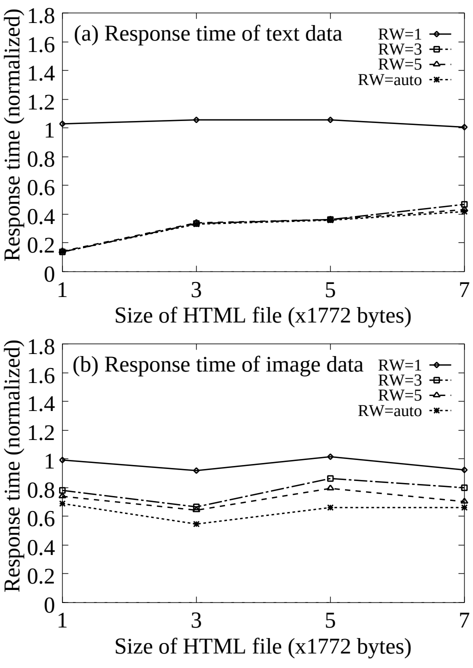

Figure 1. The effect of RW for various sizes of HTML files (1, 3, 5 and 7 times the original size).

On the other hand, the results when RW=5 and when RW=auto are about the same.

Figure 1(b) shows that the response times of image data are improved. This could be because our policy also gives favorable treatment to any server process handling an Image file that has waited for a long time in the wait state, over other processes including the computation intensive background process.

However, we did not notice any difference in the degree of improvement between the results when RW=auto and the results when RW=5 in which our mechanism produces the best improvement among fixed values of RW. So, we decided to measure the server performance when the size of the HTML files is increased more, which results in an increase in the number of times a server process handling an HTML file changes between run state and wait state. And our results show that the response times of text data when RW=auto are better than when RW=5, after the sizes of HTML files are more than 40 times the original size at which the number of disk accesses whether the file are stored in consecutive blocks or not is more than the RW value of 5.Therefore, it can be said that RW is accurately predicted and set by our mechanism, and setting RW based on PFS reflects the behavior of server processes.

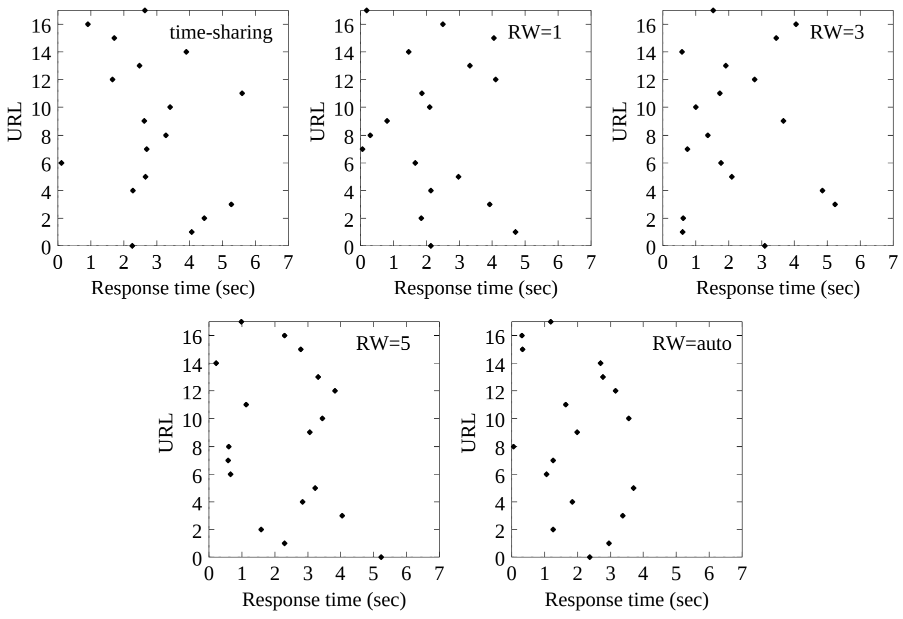

Figure 2. The distribution of response times of text data when the size of HTML file is the original size.

5.2.2. Response Time During A Number of Simultaneous Requests

In this section, all experiments were designed to examine the performance of the server when it is accessed by many requests simultaneously. All tests were conducted using three client machines and the server machine in a 10 Mbps Ethernet environment. Generally, HTML files are small [13],[14], so varying the sizes of HTML files in the same range as in the first experiment should be sufficient for the rest of the experiments.

Second experiment. In this experiment, we measured the response times when HTML files and Image file are respectively varied and fixed in the same way as in the first experiment. In order to see the effect of our predicting RW method clearly, we ran three browsers from each of the three machines simultaneously; this number can cause bottleneck of the CPU, which is indicated by the non-zero length of the ready queue, during the short period of simultaneous accesses [6]. And, we set all of the browsers to access the WWW server simultaneous every 30 seconds while SLP was fixed at 20 seconds, so that SLP would be predicted 100% correctly. Note that all browsers accessed 18 unique URLs (Uniform Resource Locator), all of which have the same content. The experimental results are shown in Figures 2 to 5.

Figure 2 shows the response times of text data when the size of HTML file is the original size. For comparison, we also measured the performance with a conventional time-sharing mechanism. Figure 2 plots the URLs in numerical sequence on the y-axis against the response time of text data to a request in seconds. This figure shows that the distributions are skew toward the small values when using our mechanism.

In order to make the experimental results easier to understand and discuss, we put all the data shown in Figure 2 into one graph as shown in Figure 3(a). Figure 3(a) illustrates the minimum, the maximum, the mean, the median values, and the range or the distribution of the response times of text data to a request in seconds. Note that the median value can be skew if we calculated it from the response times of text data, each of which is an average of 5 trial times of time1. So we calculated it directly from the raw data, that is, we found the mean of the two middle values from 90 trial times of time1 arranged in order of magnitude. Figure 3(a) shows that the mean response times of text data when RW=1, 3, 5 and auto are improved 22.9%, 21.0%, 19.0% and 31.8% respectively, while the median response times are improved 20.4%, 14.4%, 15.3% and 36.5% respectively. These figures are calculated by $\frac{a-b}{a} \times 100\%$, where a and b are the mean (or the median) values when not using and using our mechanism respectively. Besides, the maximum response times are also pressed down when using our mechanism, especially when RW=auto.

The experimental results pertaining to image data shown in Figure 3(b) are plotted in the same way. This

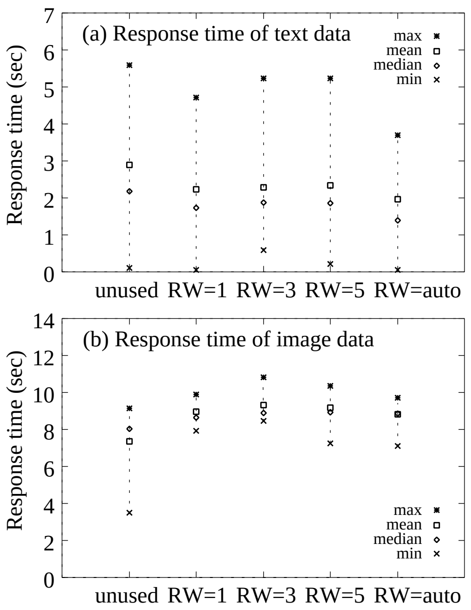

Figure 3. The response times when the size of HTML file is the original size.

figure shows that the mean and the median response times when using our mechanism are not fast as when not using ours. This is expected and is due to our policy of giving favorable treatment to processes handling HTML files over all other processes including server processes handling Image files.

In order to compare and discuss each set of results, we plotted the minimum, the maximum, the mean and the median values of response times for each size of the HTML files as shown in Figures 4 and 5. Note that each response time on the y-axis is normalized by its response time for a conventional time-sharing mechanism.

Figure 4 shows that the maximum response times of text data are pinned down when using our mechanism even though the minimum values are higher in some cases. In addition, the mean and the median values for each size of HTML files become smaller. The results imply that the service of the WWW server will be more equally distributed among users and users will perceive a faster response when using our mechanism. Also, our mechanism produces the best improvement for every case when RW=auto.

Figure 5 shows that the affect of unfairness due to our policy of giving favorable treatment to server processes handling HTML files on the max, the mean and the median response times of image data are relatively small. This means that the server processes handling Image files pay little penalty under our scheduling mechanism.

Third experiment. This experiment measured the performance of the server in a more realistic situation, that is, when it is accessed randomly by multiple requests at the same time. This test was conducted in the same environment as in the second experiment, except the browsers were set to access the WWW server randomly but at the same time. In this experiment, not only RW but also SLP were set automatically every 500 milliseconds due to the random accesses from browsers. Also, our previous work [6] has already showed that SLP is effectively predicted and updated by our mechanism based on the execution behavior of the WWW server. Therefore, the affect on the results of automatically setting SLP based on PFS can be thought as small. The experimental results are shown in Figures 6 and 7.

Figures 6 and 7 show that the trend of the results is similar to those in the previous experiment. That is, in Figure 6, the maximum response times of text data are pinned down when using our mechanism even though the minimum values are higher in some cases. And, Figure 7 shows that the affect of unfairness due to our policy on the max, the mean and the median response times of image data are relatively small. Also, in this experiment our mechanism generally produces the best improvement when RW=auto.

However, unlike the results in Figure 4(d), the median values in Figure 6(d) when RW=1 and 3 and the size of HTML file is 3 times the original size are bigger than when not using our mechanism. In this case, we decided to look at each data (i.e., each time1 of the 90 trial times) in detail, besides the two middle values we used to calculate the median. And we found that the percentage of the number of time1's for RW=1 and 3 that are faster than those at the same order when not using our mechanism are respectively 84% and 75%. However, those numbers used to calculate the median response times for both RW=1 and 3 are unfortunately not in this large percentage. As a result, the median response times for RW=1 and 3 in this case are higher while their mean response times shown in Figure 6(c) are smaller. Also, for all sizes of the HTML files, the percentage of the number of time1's when using our mechanism that are faster than those when not using our mechanism, are in the range of 74% to 100% for this experiment and in the range of 66% to 98% for the previous experiment.

Therefore, by using our scheduling mechanism the user requirement, which is the enhancement of response time during periods of high demand, can be satisfied without making any changes to the existing WWW server software.

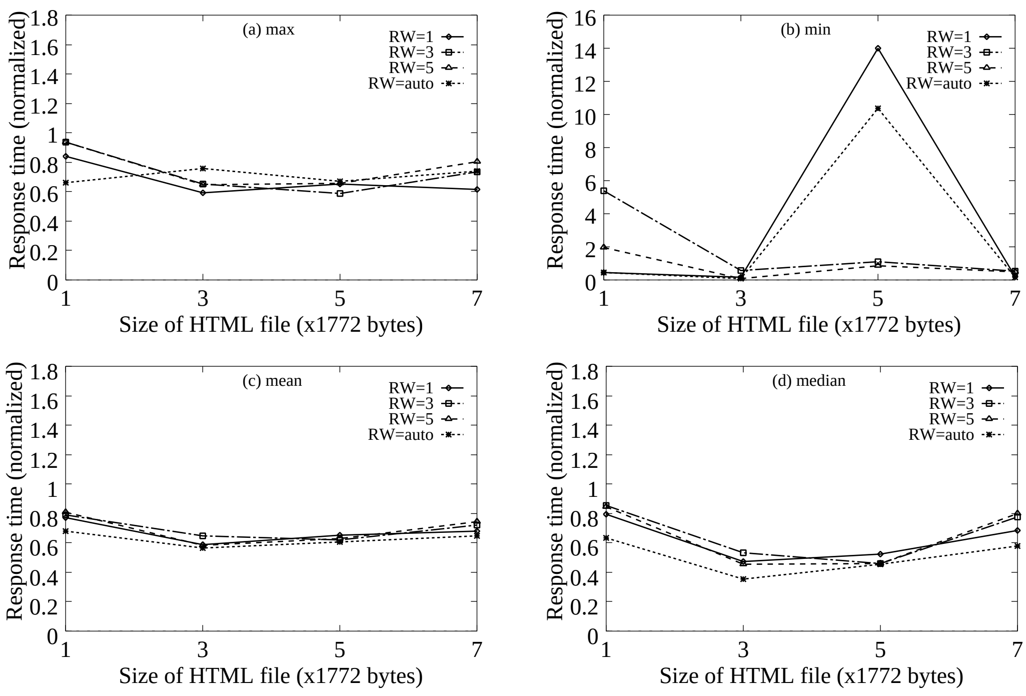

Figure 4. The effect of our mechanism on response times of text data, when browsers access the server every 30 seconds at the same time.

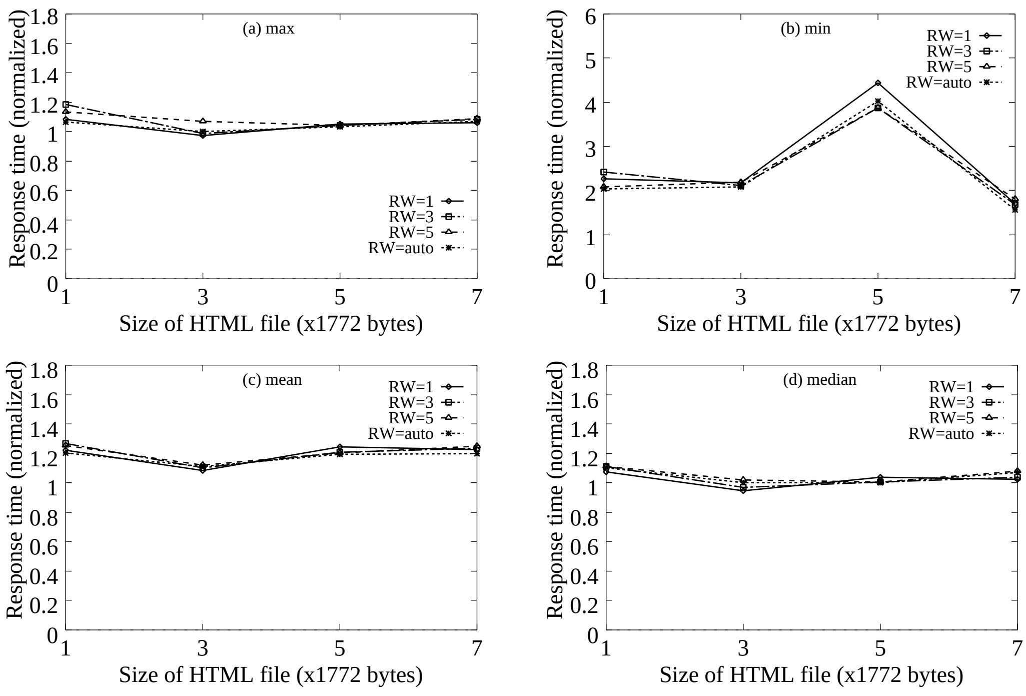

Figure 5. The effect of our mechanism on response times of image data, when browsers access the server every 30 seconds at the same time.

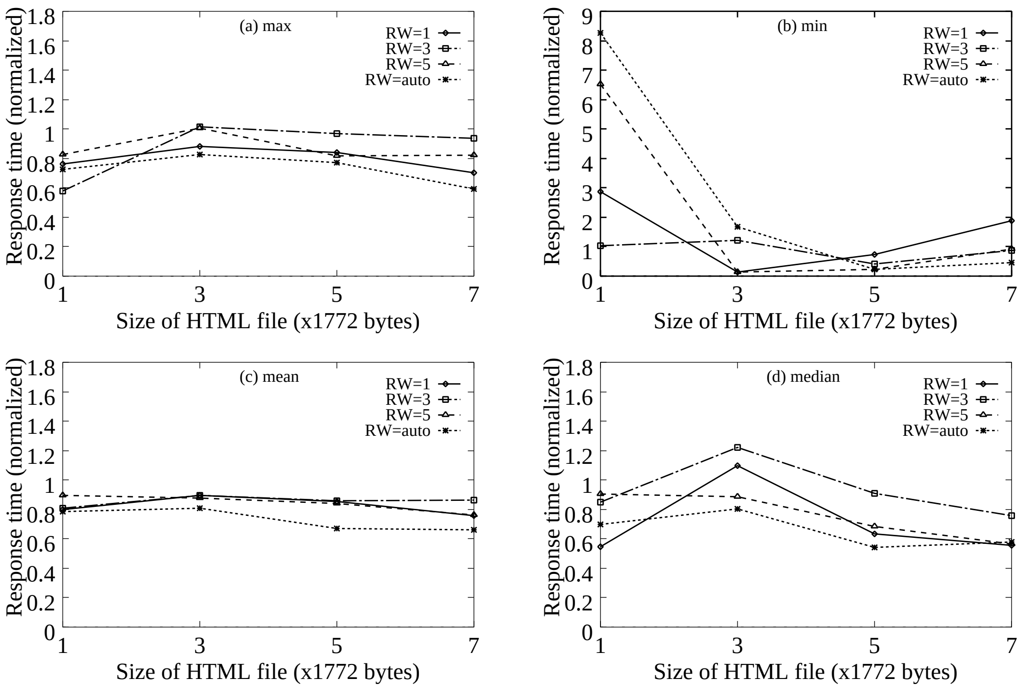

Figure 6. The effect of our mechanism on response times of text data, when browsers access the server randomly at the same time.

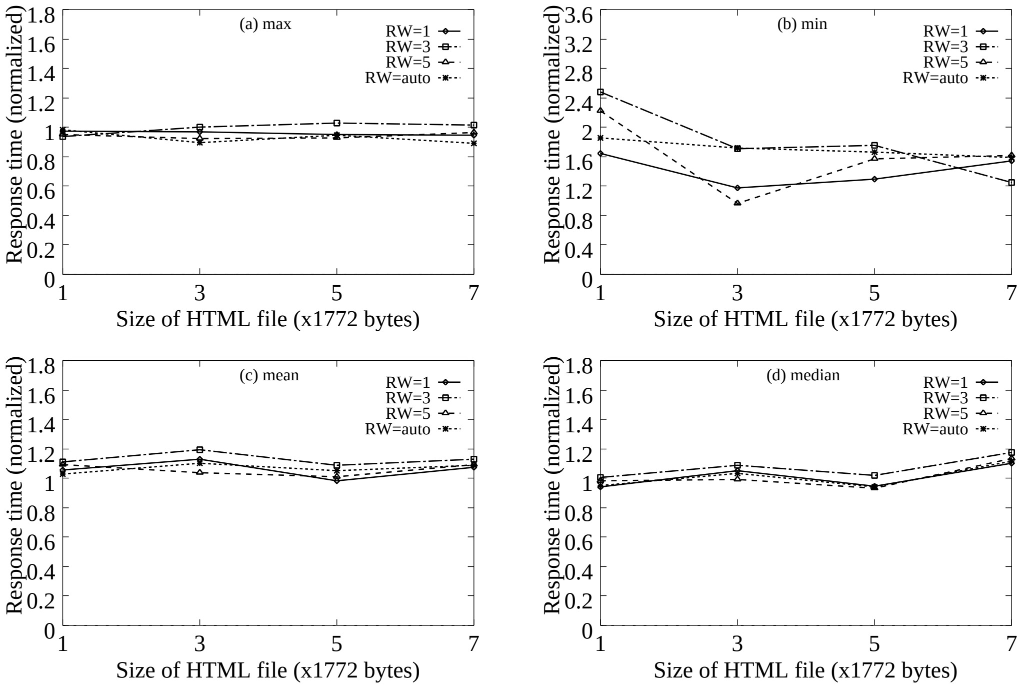

Figure 7. The effect of our mechanism on response times of image data, when browsers access the server randomly at the same time.

6. Conclusions

This paper described how we predicted and automatically updated scheduling parameter RW, which is one of the two scheduling parameters used to determine which processes are server processes handling HTML file requests, based on the observed execution behavior of a WWW server called PFS. We also verified the effectiveness of our proposed method by evaluating the performance of the WWW server experimentally when it is accessed by many requests at the same time.

Our experimental results when browsers access the server periodically at the same time show that the maximum response times of text data when using our scheduling mechanism are pinned down compared with those when not using our scheduling mechanism, even though the minimum values are higher in some cases. Besides, the mean response times become smaller and the number of smaller response times increases. These results imply that the service of the WWW server will be more equally distributed among users and users will perceive a faster response when using our scheduling mechanism. Also, our scheduling mechanism generally produces the best improvement when RW is predicted and updated automatically based on PFS. Moreover, the effect of unfairness due to our policy of giving favorable treatment to server processes handling HTML files on the response times of other types of data, in this case, image data, are relatively small. In a more realistic situation, that is, when the WWW server is accessed randomly by multiple requests at the same time, the experimental results also show the same trend. Therefore, by using our scheduling mechanism the user requirement, which is the enhancement of response time during periods of high demand, can be satisfied without making any changes to the existing WWW server software.

Future work will evaluate the usefulness of our idea using other existing software applications, and build a model of the software application using the evaluated results, and generalize our scheduling mechanism to support the evolution of the software applications we model.

Acknowledgements

We would like to thank James Michael Perry for his assistance in proofreading this paper.

References

[1] N. J. Ford and M. Woodroffe. *Introducing Software Engineering*. Prentice-Hall, 1994.

[2] R. S. Pressman. *Software Engineering: A Practitioner's Approach, 3rd ed.*. McGrawHill, 1992.

[3] I. Sommerville. *Software Engineering, 4th ed.*. AddisonWesley, 1992.

[4] C. F. Kemerer. A longitudinal empirical analysis of software evolution. In *Proc. of the 1998 International Workshop on Principles of Software Evolution*, 1998.

[5] S. Suranauwarat and H. Taniguchi. Process scheduling policy for a WWW server based on its contents. *Trans. IPSJ*, 40(6):2510-2522, 1999. (in Japanese)

[6] S. Suranauwarat and H. Taniguchi. Evaluation of process scheduling policy for a WWW server based on its contents. In *Proc. of the 1999 IPSJ Computer System Symposium*, 1999.

[7] S. Suranauwarat, H. Taniguchi, and K. Ushijima. Evaluation of process scheduling mechanism for a Web server based on its behavior while executing. In *Proc. of the 6th Asia Pacific Software Engineering Conference*, 1999.

[8] D. E. Comer. *Computer Networks and Internets, 2nd ed.*. Prentice-Hall,1999.

[9] A. Tanenbaum. *Computer Networks, 3rd ed.*. Prentice-Hall, 1996.

[10] M. F. Arlitt and C. L. Williamson. Internet Web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Networking*, 5(5): 631-645, 1997.

[11] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Tech. Rep. BU-CS-95-010, Computer Science Department, Boston University, 1995.

[12] C. Musciano. *Tuning Unix for Web Service*. URL:http://www.sunworld.com/swol-01-webmaster.html, 1996.

[13] M. E. Crovella and A. Bestavros. Self-similarity in world wild web traffic: Evidence and possible causes. In *Proc. of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996.

[14] J. M. Almeida, V. Almeida, and D. J. Yates. Measuring the behavior of a World-Wide Web server. Tech. Rep. BU-CS-96-025, Computer Science Department, Boston University, 1996.