# A Disk Scheduling Mechanism for a WWW Server

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*--**With the ongoing growth of the WWW (World Wide Web) has come an increase in the number of simultaneous requests servers must handle. As a result, users experience slower response times during periods of high demand. In other words, it takes a longer time for the first data to display on browsers, the text data stored in an HTML (HyperText Markup Language) file, to start displaying when servers are accessed by many requests simultaneously. This situation could be one in which it is most desirable to improve the response time and this can be achieved by scheduling resources more efficiently in operating systems. We have proposed a disk scheduling policy for improving response time of a WWW server. This policy gives preferential use of the disk drive to any process that is predicted based on its behavior to be a server process handling an HTML file request, by moving its I/O requests to the head of the I/O queue. In this paper, we implement the proposed policy and present the experimental evaluation of our disk scheduling mechanism.**

*Keywords*--**Disk scheduling, WWW server, Operating system, Behavior, Predict, Performance analysis**

## I. Introduction

The WWW (World Wide Web) has experienced a phenomenal growth and has become the most popular Internet application. It has succeeded because it gives users quick and easy access to a tremendous variety of information on remote locations. With this ongoing growth has come an increasing demand on WWW servers, which results in an increase in the number of simultaneous requests that servers must handle. As a result, users experience slower response times on the WWW sites during periods of high demand. In other words, it takes a longer time for the first data to display on browsers, the text data stored in an HTML (HyperText Markup Language) file, to start displaying when the servers are accessed by many requests simultaneously. This can cause users to get frustrated or give up waiting or even stop accessing those sites any more. Therefore, in such a situation, a server's response time has become a critical issue for improving the quality of service on the WWW.

A way to achieve the goal of improving a server's response time is to improve operating systems support for servers, especially in the area of resources allocation in which traditional operating systems still perform poorly. This is because most traditional operating systems control the allocation of the resources among processes using a fixed scheduling policy, in which the utilization of a computer system (e.g., a real-time or a time-sharing system) is a major concern rather than *contents* or *behavior of processes*. For example, in a time-sharing system, several processes may be ready to run if they could use the CPU if it were available. Since only one process can be running at a time, the rest will have to wait in the queue until the CPU is free and rescheduled on a round-robin basis. In addition, several processes may be generating I/O requests. If processes make I/O requests faster than they can be serviced, waiting queues buildup for each I/O device. In the case of a disk drive, traditional operating systems normally enter I/O requests into the queue in such a way that the requests will be serviced with minimum mechanical motion [1], [2], since it is likely to improve the overall performance, however, possibly at the expense of individual requests. In the case of WWW servers, the individual requests, especially the HTML file requests, are important. Therefore, with traditional operating systems, when the number of server processes needed to handle the simultaneous requests increases, if a server process handling an HTML file request or its I/O request is put at the end of the queue, then it takes a longer time for the text data to show up on browsers.

Therefore, we proposed a process' behavior-based scheduling policy for improving the response time of a WWW server [3]. This policy gives preferential use of the resources such as a CPU resource (or I/O devices) to any process that is predicted based on its behavior to be a server process handling an HTML file request, by moving it (or its I/O requests) to the head of the waiting queue. This allows each user to get an HTML file faster during periods of high demand, while other types of files (e.g., an Image file), which are embedded in the HTML file by reference, are coming in. In other words, this allows each user to view text and general layout of a WWW page in a timely manner during periods of high demand while other types of data such as an image are coming in, and also to allow the user to stop loading if the page is not sufficiently interesting to warrant waiting.

We have already implemented and evaluated the proposed policy focused on the allocation of a CPU resource [4], [5]. And our evaluations show that the mean response time of the WWW server is improved as much as 22% when using our CPU scheduling mechanism compared with that when using the conventional priority-based time-sharing mechanism. In this paper, we implement the proposed policy focused on the allocation of an I/O device as a disk drive, and present the experimental evaluation of our disk scheduling mechanism.

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 policy. Section 4 give important aspects of implementation. Section 5 describes our experiments and explains the results we obtained. Section 6 discusses related work. Section 7 offers our conclusion and future work.

## II. The WWW

The WWW is based on the client-server model [6], [7]. That is, users access WWW pages provided by WWW servers via WWW clients, mainly browsers. WWW pages are written in 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 files and Image files account for more than 90% of the total requests to a server [8], [9]. 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, Internet Explorer, or Mosaic. 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 1.2.5 [10], the pre-forking model server. In this model, a *master* process pre-forks 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.

## III. Overview

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 during periods of high demand. In other words, 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 start viewing a WWW page. Hence, in such a situation, our scheduling goal is to display the text data for the user to view as soon as possible while other types of data such as an image are coming 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 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., processes in the *ready* state) are placed on a ready queue, whereas processes that are blocked awaiting an event (i.e., processes in the *wait* state) are located on a sleep queue associated with the event. When a process is blocked awaiting an event to happen such as the completion of its I/O request, if the desired I/O device (e.g., a disk drive) is available, the request can be serviced immediately. If that device is being used by any other process, then the request will be put into the I/O queue for that device. By reducing the time a process waits for the CPU to become available in the ready queue or the time its I/O requests wait to be serviced in the I/O queue, we can reduce the processing time of a process, which results in an enhanced response time if that process is a server process handling an HTML file request. For this reason, we proposed the scheduling policy that when a CPU (or an I/O device) becomes bottlenecked, any server process handling an HTML file request (or any of its I/O requests) will be moved to the head of the ready queue (or the I/O queue for that device). Note that the bottleneck mentioned in this paper is the situation in which the resources are being used and there is more than one process waiting to use the resources.

## IV. Implementation

When we discussed the scheduling mechanism that implements the proposed scheduling policy focused on the bottleneck of a disk drive, we had *two* problems: *how to detect which processes are server processes handling HTML file requests*, and *how to operate the I/O queue*. Since the first problem is also one of the problems we had when we implemented the proposed policy focused on the bottleneck of a CPU resource [3], we will briefly describe it here in order to provide sufficient understanding to the rest of the paper.

**To deal with the first problem**, we first need to know what a server process handling an HTML file request is like. So, we logged the execution behavior of a WWW server in terms of process identifier, process state and time. And based on this log, we created the predicted execution behavior called *PFS (Program Flow Sequence)* for each server process. PFS is a sequence of entries describing process state and time spent. We analyzed the behavior of server processes based on PFS and found that a server process handling an HTML file request is a process that waits for a request from a browser, and the time it waits for a request is relatively long compared with the time waiting for other kinds of events to happen (e.g., the completion of an I/O request) in the wait state. Also, after accepting a request, it tends to change between run state and wait state a number of times. The number of changes is proportional to the size of the file it handles, i.e., an HTML file (which is usually smaller than an Image file). In other words, any server process handling an HTML file request 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 request (characteristic 2).

Next, we introduced two parameters into our mechanism in order to determine which processes have the above two characteristics (i.e., to determine which processes are server processes handling HTML file requests): 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 request. By these two parameters, we can detect which process appears to be a server process handling an HTML file request. SLP and RW are automatically predicted and updated every time period based on the predicted execution behavior of each server process, i.e., PFS of each server process. We note that we cannot fix SLP and RW at some values due to random accesses from users (in the case of SLP), and the changing execution behavior of the server and the size of the HTML files it handles (in the case of RW). Also, PFS is created and updated every time period based on the log we collected when the WWW server is running. As mentioned, a log is a sequence of entries describing process identifier, process state and time.

**To deal with the second problem**, our disk scheduling mechanism gives preferential use of the disk drive to any process that is predicted to be a server process handling an HTML file request, by moving its I/O requests to the head of the I/O queue. In other words, our mechanism puts any I/O request from any process that has characteristic 1 at the head of the I/O queue; and when that process loses characteristic 2, its I/O requests will be scheduled normally, i.e., its I/O requests will be put into the I/O queue using a routine provided by the operating system, which in our case is the *disksort* routine. Disksort enters I/O requests into the queue in a cyclic, ascending, cylinder order as described below.

An I/O queue is made up of one or two lists of requests ordered by cylinder number. The request at the front of the first list indicates the current position of the drive. If a second list is present, it is made up of requests that lie before the current position. Each new request is sorted into either the first or the second list, according to the request's location. When the heads reach the end of the first list, the drive begins servicing the other list. Figure 1(a) shows an example of an I/O queue with requests for I/O to blocks on cylinders 75, 30 and 120 when the request at cylinder 100 is being serviced. For comparison, Figures 1(b) and (c) show respectively how disksort and our mechanism enter the requests from a server process handling an HTML file request on cylinders 140 and 80, in addition to the requests mentioned in Fig. 1(a). Note that in the case of the operating system in which we implemented our disk scheduling mechanism (BSD/OS 2.1), the request at the head of the queue is the one that is being serviced. So, we decided to enter the request of any server process handling an HTML file request right after the one at the head. Also, when there is more than one request generated by server processes handling HTML file requests, then they will be put into the queue on a first-come-first-served basis.

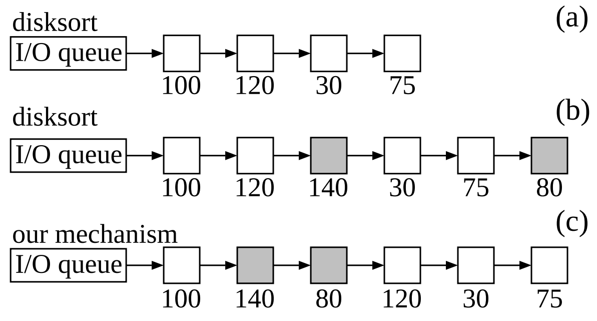

Fig. 1. Examples of how disksort and our mechanism enter I/O requests into the queue.

## V. Experimental Evaluation

In this section, we present three experiments designed to evaluate the effectiveness of our disk scheduling mechanism, which is implemented as a modification to the BSD/OS 2.1 kernel. In experiment 1, we used an I/O intensive program (test program) to verify that our scheduling mechanism works as expected, i.e., when the disk drive is bottlenecked the processing time of the test program can be reduced by using our disk scheduling mechanism due to its policy of moving the I/O requests generated by the process running on behalf of the test program to the head of the I/O queue. In experiment 2, we examined the performance of the WWW server in a simple case in which the WWW server was accessed by a single browser. In order to clearly see the effect of our disk scheduling mechanism, we ran three I/O intensive background processes that bottlenecked the disk drive of the server machine 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. The good result obtained in experiment 2 motivated us to perform experiment 3 in order to examine the performance of the WWW server in a more realistic case in which the WWW server was accessed by multiple browsers randomly at the same time, and no I/O intensive background process coexisted, and SLP and RW were predicted/updated automatically every 500 milliseconds due to random accesses from browsers and the changing execution behavior of the server.

### A. *Experimental 1*

#### 1) *Experimental Setup*

Our test program is a program that reads a text file which is

8,000 bytes. In this experiment, we varied the number of coexisting I/O intensive background processes from 1 to 3, and we measured the processing time or the time the test program took to read the text file when using disksort and when using our disk scheduling mechanism. Besides, we measured the processing time when the content of the I/O queue was logged while using our scheduling mechanism. For comparison, we also show the result when there was no coexisting I/O intensive background process. Each I/O intensive background process looped the file-read of 20 different text files.

Experiment 1 was run on a 233 MHz AMD-K6 with 64 MB of memory, running our modified version of BSD/OS 2.1. Also, the experiment was conducted in single user mode, and the operating system's I/O buffer cache was disabled during the experiment in order to clearly see the effect of our disk scheduling mechanism.

#### 2) *Experimental Results*

Figure 2 shows the average of 16 trial times of the time the test program took to read the text file when using disksort and when using our disk scheduling mechanism.

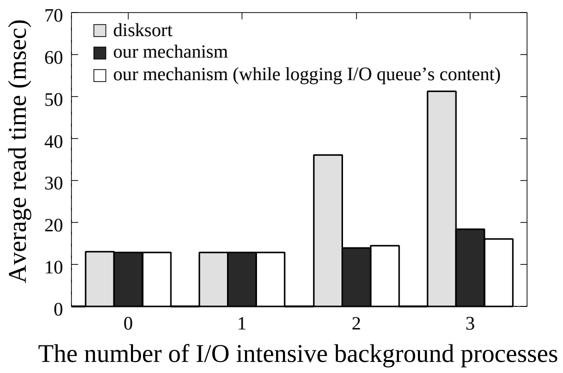

Fig. 2. The effect of our disk scheduling mechanism when using a test program.

The experimental results are described below.

- In the case of using our disk scheduling mechanism, the results when the content of the I/O queue was logged and when it was not logged show the same trend.
- In the case of using disksort, the average time the test program took to read the text file when there was no coexisting I/O intensive background process is the same as when there was a coexisting I/O intensive background process. Accordingly, the bottleneck of the disk drive does not occur when there is a coexisting I/O intensive background process. As a consequence, we cannot notice the effect of our scheduling mechanism when there is only one coexisting I/O intensive background process, due to its policy of moving any server process handling an HTML file request to the head of the I/O queue when the disk drive becomes bottlenecked.
- When the number of coexisting I/O intensive background processes is more than two, the effect of our disk scheduling mechanism is noticeable. That is, the time taken to read the text file is decreased when using our scheduling mechanism. This improvement also points out that the bottleneck of the disk drive occurs when there is more than two coexisting I/O intensive background processes. In addition, the results agree with the content of the I/O queue we logged, i.e., the length of the I/O is not zero and the I/O requests generated by the process running on behalf of the test program is moved to the head of the queue.

The above results show that when the disk is bottlenecked the time the test program takes to read a file can be reduced by using our disk scheduling mechanism.

### B. *Experiment 2*

#### 1) *Experimental Setup*

The software used for the WWW server and the browser in this experiment was Apache 1.2.5 and Netscape Navigator 3.04 respectively. The WWW server ran on a personal computer with a 233 MHz AMD-K6 processor and 64 MB of memory, running our modified version of BSD/OS 2.1. On the other hand, the browser ran on a personal computer with a 200 MHz Intel Pentium Pro processor and 64 MB of memory, running on BSD/OS 2.1. The server machine and the client machine were connected by a private 10 Mb/s Ethernet. Also, the experiment was conducted in single user mode, and the operating system's I/O buffer cache in the server machine and each browser's cache were disabled during the experiment in order to clearly see the effect of our disk scheduling mechanism.

In this experiment, the browser accessed the WWW server every 30 seconds when the WWW server coexisted with three I/O intensive background processes and SLP was fixed at 20 seconds. Such a situation is the best for our scheduling mechanism, since the disk drive of the server machine is bottlenecked caused by the three coexisting I/O intensive background processes and any access from the browser is set in such a way that SLP would be predicted 100% correctly. In this experiment in which we varied RW in the range from 1 to 10, we measured the 5 trial times of *time1* and *time2*. 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 time of text data and response time of image data respectively. Note that, for each trial, the browser accessed the same URL (Uniform Resource Locator), an HTML file (1,772 bytes) and an Image file (43,770 bytes).

#### 2) *Experimental Results*

Figure 3 shows experimental results plotted with the response time on the y-axis normalized by the response time when using disksort. This figure shows that the response time of text data for each RW is better than that when using disksort, as expected. This is because the coexisting I/O intensive background processes always caused the disk drive to become bottlenecked and the WWW server processes waiting for

HTML file requests from browsers were always in the wait state at least 30 seconds which was more than SLP (20 seconds).

Figure 3 also 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 request that has waited for a long time (i.e., waited more than SLP of 20 seconds) in the wait state, over other processes including the coexisting I/O intensive background processes.

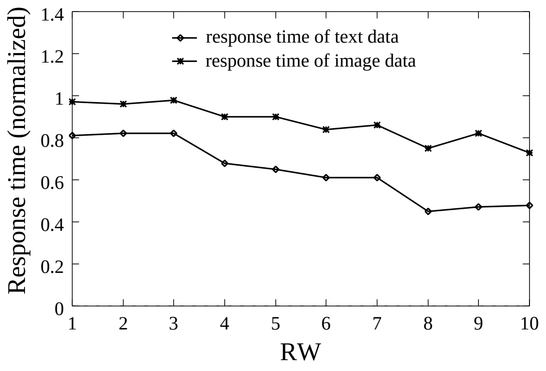

Fig. 3. The experimental results in a simple case in which the WWW server is accessed by a single browser.

### C. Experiment 3

#### 1) Experimental Setup

The software and the hardware used in this experiment were the same as those in experiment 2 except that the number of client machines was increased to three. All the client machines had the same specification as that in experiment 2 and were also connected to the server machine by a private 10 Mb/s Ethernet.

In this experiment, we set three browsers from each of the three client machines to access the WWW server randomly at the same time. According to the content of the I/O queue we logged during the pre-experiment, the number of simultaneous accesses in this experiment can cause bottleneck of the disk drive, which is indicated by the non-zero length of the I/O queue, during the short period of simultaneous accesses. Also, all browsers accessed 18 unique URLs all of which have the same content as that in experiment 2, that is, an HTML file (1,772 bytes) and an Image file (43,770 bytes). For each URL, we measured the 5 trial times of time1 and time2, in other words, we measured response time of text data and response time of image data. During the experiment, SLP and RW were automatically predicted and updated based on PFS every 500 milliseconds. And our previous works [4], [5] have already showed that SLP and RW are effectively predicted and updated by our scheduling mechanism.

#### 2) Experimental Results

Figure 4(a) illustrates the maximum, the minimum, the mean, and the median response times of text data from 18 URLs. Note that the median values are calculated directly from time1, since it can be skewed if we calculate it from the response times of text data, each of which is the average of the 5 trial times of time1. For comparison, we also show the results when the I/O requests were scheduled using disksort. Also, the results for image data shown in Fig. 4(b) are plotted in the same way.

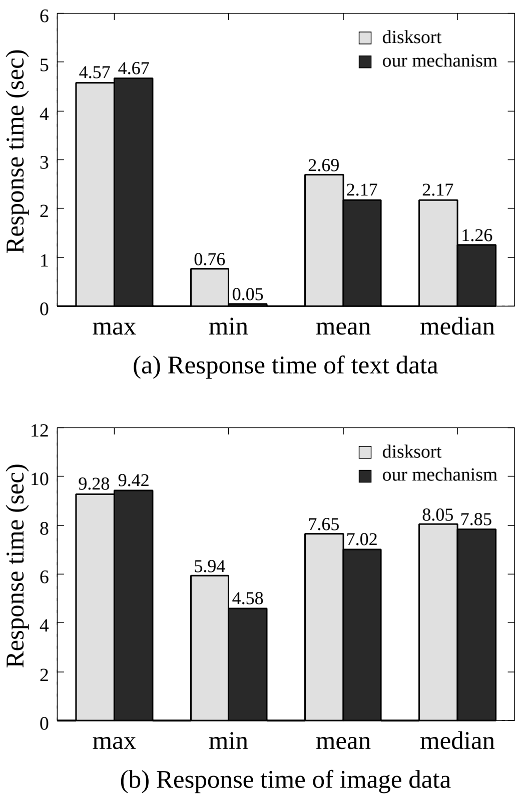

Fig. 4. The experimental results in a more realistic case in which the WWW server is accessed by multiple browsers randomly at the same time.

Figure 4(a) shows that the minimum, the mean, and the median response times of text data are improved 93%, 19% and 42% respectively, while the maximum response times get worse a little bit (only 2%). These figures are calculated by $\frac{a-b}{a} \times 100\%$ where $a$ and $b$ are the maximum (or the minimum or the mean or the median) response times of text data when using disksort and when using our scheduling mechanism respectively. According to the result, our disk scheduling mechanism produces a good improvement for response time of text data. In other words, by using our disk scheduling mechanism the time from requesting a WWW page until text data starts displaying is reduced.

However, the price to be paid for reducing the time from requesting a WWW page until text data starts displaying is that we reduce the fairness of the system, which consequently affects the amount of time it takes from requesting a WWW page until image data displays completely. If this effect is small, it is acceptable. For example, it is acceptable if the image data displays completely while the users are reading the text data that displays faster. On the other hand, if the effect on the response time of image data is big, then users might get frustrated and not wait for the whole page to display completely. Therefore, care must be taken to ensure that the resulting unfairness does not outweigh the performance gains obtained. And our result in Fig. 4(b) shows that the worst or the maximum response time of image data when using our mechanism is only 1% slower than when using disksort. According to the result, response time of image data pays a small penalty under our scheduling mechanism.

Therefore, any WWW server that experiences a lot of simultaneous accesses from users would benefit from our disk scheduling mechanism.

## VI. Related Work

The work described in this paper relates mainly to the area of disk scheduling in operating systems.

The simplest form of disk scheduling is FCFS (First Come First Served), that schedules requests in the order of their arrival. Since the access schedule thus derived is independent of the relative positions of the requested data on disk, FCFS scheduling can incur significant seek time (the time for disk arm to move the read-write heads to the cylinder containing the desired sector) and rotational latency (the additional time waiting for the disk to rotate the desired sector to the disk head). Therefore, many scheduling policies concentrated on optimizing seek time (*seek optimization*) and rotational latency (*rotational optimization*) have been proposed in order to achieve higher performance.

### *A. Seek Optimization*

The SSTF (Shortest Seek Time First) policy [1], [2] chooses the next request to service by selecting the pending request that will incur the shortest seek time. It is usually infeasible to predict exact seek times, but SSTF may be closely approximated by using seek distances. SSTF results in better throughput rates than FCFS, and mean response time tends to be lower for moderate loads. One significant drawback is that higher variances occur on response times because of the discrimination against the outermost and innermost tracks; in the extreme, starvation of requests far from the read-write heads could occur.

The SCAN policy [1], [2] was developed to overcome the discrimination and high variance in response times of SSTF. This policy operates like SSTF except that it chooses the request that results in the shortest seek distance in a *preferred directions*, i.e., inward or outward. It only changes direction when it reaches the innermost or outermost cylinder.

The C-SCAN (Circular SCAN) policy [1], [2], a variant of SCAN, replaces the bidirectional scan with a single direction of disk arm travel. When the arm reaches the last cylinder, it immediately returns to the first cylinder without servicing any requests on the return trip. C-SCAN treats each cylinder equally, rather than favoring the center cylinders. The LOOK policy [2], [11], another SCAN variation, changes scanning direction when no more requests are pending in the current direction. C-SCAN and LOOK can be combined, resulting in the C-LOOK policy [2], [11] which the disksort routine in BSD/OS 2.1 is implemented based on.

The VSCAN(R) policy creates a continuum of policies between SSTF and LOOK. It adds a penalty, which is dependent on the parameter R and the seek distance, whenever it changes direction, VSCAN(0.0) is equivalent to SSTF, and VSCAN(1.0) reduces to LOOK.

### *B. Rotational Optimization*

Paralleling the SSTF strategy of seek optimization is the SLTF (Shortest Latency Time First) strategy for rotational optimization. Once the disk arm arrives at a particular cylinder, there may be many requests pending on the various tracks of that cylinder. The SLTF strategy examines all these requests and services the one with the shortest rotational delay first.

The above policies attempt to service I/O requests with the minimum mechanical motion, but are less concerned about each request individually, which is what our policy does. As a consequence of using the above policies, when a WWW server is accessed by a lot of users simultaneously, the likelihood that an I/O request generated by any server process handling an HTML file request will be put at the end of the queue is high, which results in users experiencing a slower response.

## VII. Conclusion

This paper examined the benefit of the proposed scheduling policy that controls the allocation of a disk drive based on the behavior of WWW server processes rather than based on a fixed policy used in traditional operating systems, in which the utilization of a computer system such as a real-time or a time-sharing system is a major concern. And our experimental result when the WWW server is accessed randomly by multiple requests at the same time show that by using our disk scheduling mechanism the response time, the time from requesting a WWW page until text data starts displaying, can be reduced. To be more specific, the mean and the median response time are improved 19% and 42% respectively. Moreover, the effect of unfairness due to our policy of giving favorable treatment to server processes handling HTML file requests on the response times of other types of data, which in our case is image data, are relatively small. Therefore, any WWW server that experiences a lot of simultaneous requests from users would benefit from our disk scheduling mechanism.

Future work will measure the performance of a WWW server when the operating system’s I/O buffer cache in the server machine and each browser’s cache are enabled, and also when both CPU and I/O scheduling mechanisms are used at the same time.

## Acknowledgment

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

## REFERENCES

[1] H. Deitel, *An Introduction to Operating Systems (2nd ed.)*, Addison-Wesley, 1990.

[2] Silberschatz and P. Galvin, *Operating System Concepts (5th ed.)*, John Wiley & Sons, 1997.

[3] S. Suranauwarat and H. Taniguchi, “Process scheduling policy for a WWW server based on its contents,” *IPSJ Trans.*, vol.40, no.6, pp.2510-2522, 1999. (in Japanese)

[4] S. Suranauwarat and H. Taniguchi, “Evaluation of a process scheduling policy for a WWW server based on its contents,” *IEICE Trans. Inf. & Syst.*, vol.E83-D, no.9, pp.1752-1761, 2000.

[5] S. Suranauwarat and H. Taniguchi, “Operating systems support for the evolution of software: an evaluation using WWW server software,” In *Proc. of the 2000 International Symposium on Principles of Software Evolution*, pp.292-301, 2000.

[6] D. Comer, *Computer Networks and Internets (2nd ed.)*, Prentice-Hall, 1999.

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

[8] M. Arlitt and C. Williamson, “Internet web servers: workload characterization and performance implications,” *IEEE/ACM Trans. Networking*, vol.5, no.5, pp.631-645, 1997.

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

[10] http://www.apache.org/

[11] A. Worthington, “Scheduling algorithms for modern disk drives,” In *Proc. of the 1994 Conference on Measurement and Modeling of Computer Systems*, pp.241-251, 1994.

[12] R. Geist, “A continuum of disk scheduling algorithms,” *ACM Trans. Comput. Syst.*, vol.5, no.1, pp.77-92, 1987.