Providing Differentiated Levels of Service in Web Content Hosting

Jussara Almeida, Mihaela Dabu, Anand Manikutty and Pei Cao
Computer Sciences Department
University of Wisconsin-Madison
{jussara,dabu,manikuti,cao}@cs.wisc.edu

Abstract

Web content hosting, in which a Web server stores and provides Web access to documents for different customers, is becoming increasingly common. Due to the variety of customers (corporate, individuals, etc.), providing differentiated levels of service is often an important issue for the hosts. Most server implementations, however, are not structured to service requests based on different levels of quality of service (QoS). This paper presents our attempts at augmenting a popular server implementation with differentiated QoS features. We explore priority-based request scheduling at both user and kernel levels. We find that simple strategies such as controlling the numbers of processes can improve the response time of high-priority requests notably while preserving the system throughput. We also find that the kernel-level approach tends to penalize low-priority requests less significantly than the user-level approach, while improving the performance of high-priority requests similarly. Based on our experiments, we discuss the bottlenecks and limitations from kernel implementations that prevent the augmented server from achieving better performance.

1 Introduction

Due to the explosive growth of the Web and the ever-increasing resource demands on the servers [12, 13, 10], Web content hosting is an increasingly common practice. In Web content hosting, providers who have a large amount of resources (for example, bandwidth to the Internet, disks, processors, memory, etc.) offer to store and provide Web access to documents from institutions, companies and individuals who lack the resource or the expertise to maintain a Web server. The service is typically provided with a fee, though some servers do not charge fees for non-commercial accounts. The variety of customers means that the expectations and requirements on the quality of the hosting service differs, and the amount of money each customer is willing to and can afford to pay differs. Naturally, customers expect that the requests from their potential clients (users that access their web pages) are serviced with a quality proportional to the amount of money they pay. Thus, it is an interesting and important issue that the Web servers provide different levels of quality of service for requests to documents belonging to different customers.

However, most Web servers today do not provide support for differentiated quality of service. To do so would require the incoming requests be classified into different categories and different levels of service be applied to each category. Apache [15], one of the most used Web servers, handles incoming requests in a *first-come first-served* manner. All the requests correctly received are eventually handled, regardless of the type of requests and the load on the system. Thus, if an individual's Web page draws many hits, those hits can monopolize the resource on the server and lengthen the response time for requests to corporate clients' Web pages, even though corporate clients may have paid much higher fee than the individual.

In this paper, we investigate one way to provide differentiated quality of service: priority-based request scheduling. Our main metric for the quality of service is the latency in handling the HTTP request to the Web page (in other words, the response time to the request). We study both user-level and kernel-level approaches. In the user-level approach, Apache is modified to include a scheduler process, responsible for deciding the order in which the requests should be handled. The scheduler restricts the maximum number of concurrent processes servicing requests of each priority. In the kernel-level approach, the Linux kernel [5] is modified such that request priorities are mapped into priorities of the HTTP processes handling them.

Using the Webstone benchmark suite, we measure the effect of the approaches on request latency. We find that the user-level approach can reduce the latency of high priority requests by up to 21%, while penalizing the low priority requests by up to 206%. The penalty, however, mostly comes from our closed-queue experiments. The system's throughput is not affected by the request scheduling. While performing similarly to the user-level approach. the kernel-level approach is more robust and is capable of providing differentiated quality of service for more diverse workloads than the user-level approach.

In this paper, we limit our investigation to Web server systems only, without addressing the quality of service issues in the networking infrastructure. In other words, we assume that the order in which the request packets arrive at the server and in which the response packets are transmitted in the Internet are completely out of the control of the Web server, and the server has to try its best to improve the response time for high priority requests. Networking quality of service, including differentiated quality of service, is a rich and active research field. Obviously, a complete solution would require a combination of networking QoS and server QoS supports; investigating the combination, however, would be out of the scope of the paper. Here, we focus on the end systems, and investigate issues such as process scheduling and OS resource scheduling that are not typically addressed in the networking QoS studies.

Previous works have addressed the general problem of quality of service by using different techniques such as data prefetching [4] and image compression [11]. Other approaches [1, 8] address the problem in the context of a distributed web server. In [1], the authors developed a system that implements fast packet interposing that can be used to balance the load across a cluster of servers. In [8], the authors address the problem of load balancing in a cluster of servers by using secondary IP addresses. Variation of these approaches can be used to some degree to provide differentiated quality of service; however, they are intended for a cluster of Web servers, while we focus on a single-machine server system.

The paper is structured as follows. Section 2 describes the methodology used. Section 3 presents the design and some highlights on implementation issues. Section 4 describes the experimental environment and the workload used in the study. Section 5 presents the main latency results. Section 6 discusses limitations of this study. Finally, Section 7 summarizes the results.

2 Methodology

We study the case where there are only two levels of quality of service needed. For example, a hosting service may provide a high level of quality of service for corporate clients, who pay the hosting fees, and a low level one for individual customers, who may get the service for free. Though simple, the problem provides a good test case for any differentiated QoS mechanism.

We investigate priority-based approaches to provide differentiated quality of service, that is, requests are attached with priorities and serviced according to the priorities. Clearly, requests are assigned priorities based on which documents they are requesting, not where they come from. In other words, customers who buy the high level of quality of service want Web accesses to their documents to be fast, regardless of who is accessing the documents.

Since in current web servers, there is no difference among requests in terms of priorities, we study two approaches to modify the Web servers:

- **User-level approach:** We have modified a popular web server program, Apache, by adding a *scheduler* process, that decides the order in which requests should be serviced.

- **Kernel-level approach:** We have modified both Apache and the Linux kernel by adding two new system calls that provide a mapping from request priorities into process priorities and keep track of which processes are running at which priority level.

The *performance metric* used to evaluate the effectiveness of our approaches was the average latency (elapsed time) of servicing a request, that is, the average time taken by the server from the moment it accepts a connection until it closes the connection. We used the server latency instead of the client latency (the response time perceived by the user) in order to factor out extraneous load in the network, which was not dedicated during some of our experiments. However, although the improvements reported on Section 5 are based solely on server latency, the results are consistent with the improvements seen from a client's standpoint.

When using the Webstone workload, we categorize requests into a small number of classes, each class consisting of requests that are similar in terms of the size of the requested document. Note that a request's class has nothing to do with its priority. We uses classes of requests because, due to the heavy tail characteristic of typical web server workload, average latency for the whole population of requests has no statistical meaning. Instead, we calculate per-class average latencies, and then calculate the overall average latency using the numbers of requests in each class as weights for the individual components.

3 Design and Implementation

In this section, we first give a general introduction of the policies used to implement priority-based request scheduling. Then, we describe our user-level approach of modifying the Apache code, and describe the modification to the Linux kernel in order to map the request priorities to process priorities.

3.1 Scheduling Policies

The scheduling policy decides the order in which requests should be serviced. Scheduling policies can be preemptive or non-preemptive. We have implemented preemptive scheduling at kernel-level, and non-preemptive scheduling at user level, since at the user level we cannot interrupt a running process and block it in order to allow a new process to run.

There are two important aspects of the scheduling policy. First, upon receiving a request, the scheduling policy must decide whether to process the request immediately, or to postpone the execution (*Sleep policy*). A request may be postponed for one of the following reasons :

- if the load on the system is already high, the latencies of currently executing requests would be affected.
- if the request is of a lower priority, subsequently arriving high priority requests may get affected.

Secondly, the scheduling policy must also decide when a postponed requests must be allowed to continue (*Wakeup policy*). We allow a request to continue only in place of a completed request. When a request is completed, the scheduling policy must decide which of the postponed requests, if any, should be selected to execute in its place.

If the policy allows lower priority requests to execute in case sufficient higher priority requests are not available, the scheduling policy is said to be **work-conserving**. Otherwise it is said to be **non-work-conserving**. A work-conserving policy tries not to allow processes to block while there are waiting requests.

In our implementation, the *Sleep* and *Wakeup* policies have been implemented by using thresholds for maximum number of requests (max_proc) that can be concurrently handled in each priority level. Thus, we have a fixed number of slots for each priority level, and each arriving request must either occupy a slot (*executes*), or wait in a queue until it is allowed to execute (*blocks*).

3.1.1 Non-Work Conserving Policy

The non-work-conserving policy allows requests of a certain type to occupy only slots corresponding to the same type, and thus allows a request to execute only if the total number of requests of that type is below the threshold (*Sleep policy*). For example, let us say we have two levels of priority A and B, and assume max_proc of A and max_proc of B to be 6 and 3. If three requests of type A and four of type B arrive, three requests of each type occupy their corresponding slots and one request of type B blocks. Regarding the *Wakeup policy*, we can pick a request of only those types that have not exceeded their threshold. We arbitrate among such requests using priority and age.

3.1.2 Work Conserving Policy

The work conserving policy does not allow a slot to go empty, and allows requests to occupy slots of a different type (*Sleep policy*). In the above example, all the requests would execute: three requests of each type would occupy their slots and one request of type B occupies a slot corresponding to A. Subsequently, suppose three requests of type A arrive. In such scenario, two of them will execute. The third one executes if the policy is pre-emptive, otherwise it blocks (*priority inversion*). We have implemented pre-emptive scheduling at kernel level, and non-preemptive scheduling at user level. The *Wakeup* policy is similar to that of the non-work conserving policy.

3.2 User-Level Approach

We implement the scheduling policies at user level by modifying the Apache Web server program. Apache is a multi-process server where a master process spawns a number of child processes that are responsible for handling the incoming HTTP requests. We changed the master process to spawn a separate Scheduler process as well. The Scheduler is responsible for executing the *Sleep* and *Wakeup* policies. When a process receives a request, it parses the incoming URL to determine the file requested. It then obtains *scheduling information* of the request (basically a priority value, in our current implementation) required by the Scheduler. To determine the priority, it first determines the customer to which the file pertains, and then maps the customer name into a priority value. We assume that the customer name is embedded in the url.

The process then wakes up the Scheduler to execute the *Sleep* policy, and blocks, waiting for a go-ahead from the Scheduler. When the child receives a go-ahead from the Scheduler, it retrieves the file requested, completes processing the request and closes the connection. It then wakes up the Scheduler for it to execute the Wakeup policy.

The Scheduler waits for child processes to signal it to start the *Sleep* or *Wakeup* policies. When signalled to execute the *Sleep* policy, the Scheduler chooses and wakes up one child process to proceed. The Scheduler maintains a queue of postponed requests, waiting to be executed, and inserts the request into the queue if it has to be postponed. When signalled to execute the *Wakeup* policy, the Scheduler wakes up zero or one of the child processes waiting to be executed, as dictated by the policy.

The scheduling policy is implemented as a dispatcher function, which acts as the Scheduler. The child processes and the Scheduler communicate by means of a Request Board and an Event Board, implemented in shared memory. The Request Board is an array of cells, one corresponding to each child process, which is used by the child handling a request to pass scheduling information (priority) to the scheduler. The Event Board is used to inform the Scheduler of a *Sleep* or *Wakeup* event.

Thus, the Scheduler works in a tight loop, waiting for a new to process. Since only shared memory and semaphores are used, synchronization and communication are fast and the overhead of scheduling policy is kept low. The Scheduler processes *Wakeup* events before *Sleep* events, in order to free resources as soon as possible and to avoid race conditions.

Note that our implementation may create more processes than necessary in handling the Web requests. This is not a problem in our current experiments, as the network links are typically the bottlenecks. The implementation can be easily modified should the number of processes become a problem.

3.3 Kernel-Level Approach

The motivation for a kernel-level approach is that acting directly on the priorities assigned to the HTTP processes might be more effective in controlling their executions. Therefore, we use a direct mapping from the user-level request priority to a kernel-level process priority.

The kernel-level scheme is based on the instrumentation of both Apache and Linux code. Apache is changed to have each HTTP process call the kernel to record the priority of the current request it is handling. The request priority is obtained as described in the last section. The kernel is responsible for mapping this priority into the process priority (used by the kernel scheduler to decide which process should use the CPU next) and executing the *Sleep* policy to decide if the process should proceed or block. The kernel must keep track of all processes currently using the priority scheme and their current state (sleeping or handling a request), in order to avoid starvation.

When a process finishes handling a request, it calls the kernel again to release its priority and execute the *Wakeup* policy. The decision of which process should be waken up has two phases. First, the kernel decides the priority level of the process to be unblocked. This is done based on the policy used (work conserving or non-work conserving). Then, the kernel chooses the oldest process (the one that blocked the earliest) that originally was running on that priority level. The work conserving policy allows preemption which means that a lower priority process may be put to sleep so that a higher priority process can run. When choosing a process to sleep, the kernel picks the youngest process (the one that started running the latest) among all of those running on the lower priority. This guarantees fairness among processes of equal priority.

The parameters for the kernel level approach are: the number of priority levels and the description of each priority level including the priority value and the number of processes that are allowed to concurrently run on the priority. These parameters are defined in a configuration file read by Apache and passed to kernel through a system call just after the server startup. Another parameter is the priority value assigned to a process in order to make it block. This parameter, SLEEPING_PRIORITY, is defined in a kernel header file.

As we show in Section 5, for some experiments, there is a bottleneck in the network interface queue where the packets are inserted for transmission. All data packets are inserted in a single queue in a *first-in first-out* order. Therefore, we change the algorithm that inserts packets into this queue to take into account the priority of the process that is transmitting. In this way, we extend the idea of process priority to packet priority.

We implement three new system calls:

- *initialize_priority_scheme*: this system call must be called in the beginning of the server operation (`standalone_main`) and initializes all the kernel data structures that keep track of the processes that are using the scheme.

- *my_set_priority*: this system call receives as parameter the request priority, maps this priority into a process priority and executes the *Sleep* policy.

- *my_release_priority*: this routine is called just before the connection is closed and it informs the kernel that the handling of the request has finished so that it can execute the *Wakeup policy*.

To keep track of the states of all processes, the kernel maintains a table with one entry for each process. Each entry in this table contains: a pointer to the process PCB (*process control block*), the time it started to run, and its original priority. The kernel also keeps counters for the total number of processes and the number of running (non sleeping) processes on each priority. The mapping of request priority to process priority is done by a very simple hash function that returns a value between 0 and 40, that is, it returns the request priority modular 40. We did not try any other function because, as will be shown in section 5, the maximum number of processes is the most important factor to take into account when trying to improve performance of one type of requests.

3.4 Differences Between the Two Approaches

The user-level approach and the kernel-level approach implement very similar scheduling policies. However, there are three significant differences in the resultant process scheduling.

First, in the user-level approach, once a low-priority request is allowed to "go", the process runs at the same priority as those servicing high-priority requests. The kernel-level approach, on the other hand, allows the process' priority to be configured higher or lower (for example, see the configuration in Table 3.)

Second, the SLEEPING_PRIORITY in the kernel-level approach affects when a process actually blocks. In most of our experiments, we set SLEEPING_PRIORITY equal to 0. This actually does not make a process block, but reduces the time a process can use the CPU to less than the interval between timer interrupts (in Linux, this interval is 10 milliseconds). When all other processes are blocked on I/O, a process with priority 0 can assume the CPU. However, in the next timer interrupt the kernel scheduler will try to find a better (with higher priority) process to assume the CPU. This is different from the user-level implementation, which blocks the process immediately if the scheduler decides that the request should be postponed.

Finally, the kernel-level approach may preempt an already-running process if there are many high-priority requests waiting, while the user-level approach implements only non-preemptive scheduling.

The differences in process scheduling lead to differences in performance degradation of low-priority requests, as we will see later.

3.5 Measuring Latency

In order to measure the server latency, we implement a patch file to be linked with Apache code that provides routines for measuring time and logging results. Each process sums the latency for all the requests it handled in each class. Just before finishing its execution, each process writes per-class statistics - average latency and total number of requests - in a log file. Calls to the timing routines were inserted just after a connection is established and just after it is closed, in order to measure the server latency.

4 Experimental Setup

For the user-level approach, our Web server is a Sun SparcStation with two 66Mhz CPU's and 64 MB of main memory, running the Solaris operating system (version 2.4). For the kernel-level approach, we used the Linux 2.1.54 operating system on a DEC Celebris XL590 90MHz Pentium machine with 32 MB of main memory. The server software for both setups was Apache, version 1.3b2, a public domain HTTP server [15]. Our Apache server was configured to run in standalone mode. The KeepAlive option was deactivated (only one HTTP request was serviced per connection) and all other parameters are set as the default values.

To generate a WWW workload, we use WebStone [16] (version 2.0.0), an industry-standard benchmark for generating HTTP requests. WebStone is a configurable client-server benchmark that uses workload parameters and client processes to generate Web requests. This allows a server to be evaluated in a number of different ways. It makes a number of HTTP GET requests for specific pages on a Web server and measures the server performance, from a client standpoint. In order to generate load for a WWW server, client processes successively request pages and files from the server, as fast as the server can answer the requests.

For the user-level approach, Webstone client processes were spawned in 6 machines (5 client processes per machine) similar to the one used as the server hardware platform. Clients and server communicate through a dedicated 100Mbps Ethernet network. For the kernel-level approach, client processes were spawned on the same machines. However, in this case, the network connecting server and clients is a non-dedicated 10 Mbps Ethernet. In both cases, server and client machines were always dedicated for our experiments. We started two independent WebStone benchmarks, each one configured to spawn 15 client processes to send requests of only one specific type. In this way,

	Average File Size	**Largest File Size**	**Number of Files**
Workload WA	18.5 KB	115 KB	40
Workload WB	28.8 KB	2 MB	60

Table 1: Description of HTTP Workloads.

we guarantee that the number of clients sending requests to each type is always the same.

Webstone workload is defined by the number of client processes and by the configuration file that specifies the number of pages, their size and access probabilities. In our experiments, we set the number of client processes as 30 and use two different workloads, described in Table 1. The parameters that define workload WB are representative of the kinds of workload typically found in busy WWW servers [2].

5 Results

This section discusses the results obtained for both user and kernel level approaches. Recall that the performance metric is the average latency of a request as perceived by the server. We use 30 client processes and have 15 processes issue requests of type A (higher priority) and the rest issue requests of type B (lower priority). Since hardware and software platforms for user and kernel level approaches are different, we present our results separately and try to analyze the effectiveness of our schemes by comparing the (weighted) average latency for each type of request with the correspondent latencies when no scheme is used. We also use as a lower bound the average latency obtained when 15 clients send requests of only one type.

5.1 Results for the User-Level Approach

We use the workload WA to test the user-level approach. The variance of the results is within 4%.

In order to obtain differentiated performance, we find that it is necessary to restrict max_proc_B, the maximum number of processes allowed for lower priority B, to a small number. This has also been observed in the kernel level scheme. When this number is low, the availability of resources for A increases, and the contention decreases. These factors are expected to improve the performance of A.

In Figure 1, we show the average latency when we vary *max_proc_A* for the *non-work conserving* policy. For larger values of *max_proc_A*, we note that the performance of A is improved by 21%, while that of B falls by 206%, with respect to when no differentiated QoS policy is used. Thus, we can trade good performance for A with poor performance for B. We also note that if the number of processes for A is too low, the performance of A is impaired.

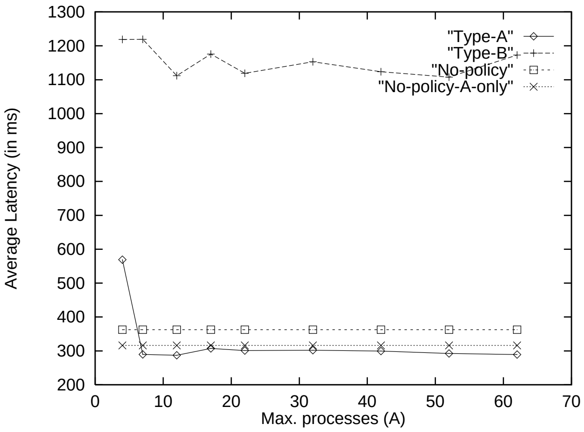

Figure 1: User-level approach: non-work conserving policy ($max\_proc\_B = 3$).

Figure 2 shows results for the same configuration but with max_proc_B set to 1. Performance of A improves by 26%, while that of B falls by 514% (note the change in scale of the graphs). The experiment shows an extreme scenario, and demonstrates that there is an appropriate number of max_proc_B below which the impact on high-priority requests is minimum.

Figure 3 shows results for the *work conserving* policy. For lower values of *max_proc_A*, the behavior is similar to the non-work-conserving scheme. For large values of *max_proc_A*, the performance degenerates to that under the original Apache server, when no differentiated QoS policy is used. When max_proc_A is equal to 35, the three curves are within 3% of each other. This is because work conserving schemes lose any notion of restriction when the thresholds are large, and waiting requests of type B can overflow into slots meant for type A. This means that in the case of large threshold values, such overflows are likely to occur and differentiated quality-of-service will not be achieved. In this case, a non-work-conserving policy works better.

We have also run experiments with the non-work conserving algorithm, keeping the value of max_proc_A

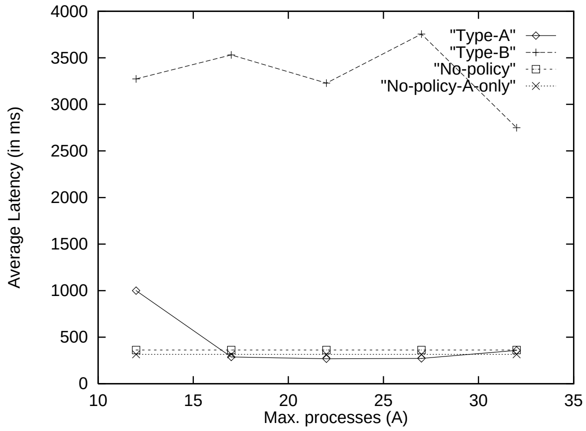

Figure 2: User-level approach: non-work conserving policy ($max\_proc\_B = 1$).

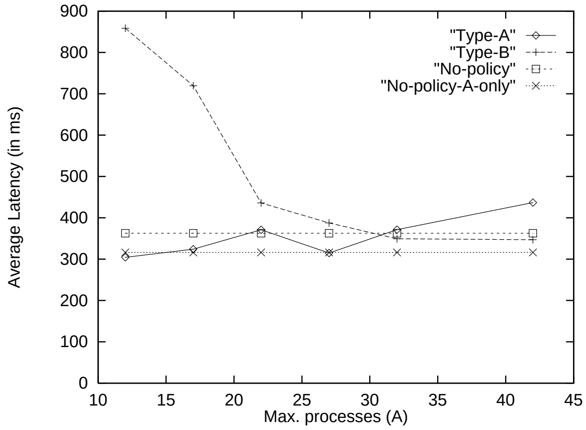

Figure 3: User-level approach: work conserving policy ($max\_proc\_B = 3$).

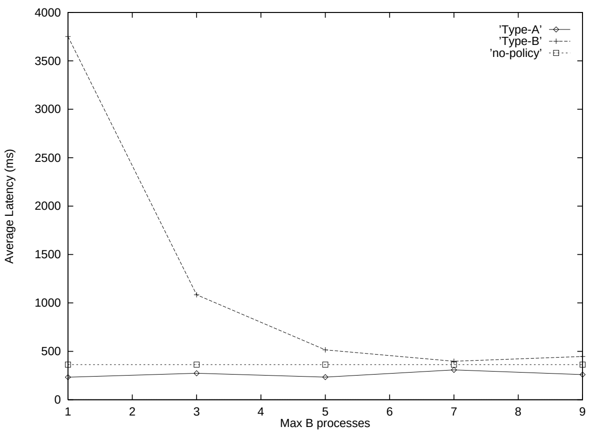

Figure 4: User-level approach: non-work conserving policy for max_proc_A = 15.

constant, namely 15 processes, and varying max_proc_B, from 1 to 9. The goal is to find the max_proc_B that keeps the performance for A better than that under no differentiation (that is, the original Apache). The results are shown in Figure 4. It can be seen that as max_proc_B increases, A's performance varies more widely. Thus the threshold for B should be max_proc_B = 5.

5.2 Results for the Kernel-Level Approach

In this section, we present results for the kernel-level approach for both workloads WA and WB. The variance of results in this section is less than 10%. Table 2 shows the average latency in the lower bound experiments: 15 clients requesting only files of type A. It also shows the results for 30 clients requesting both types of files but with the priority scheme disabled, which means that the server does not make any differentiation between the types of requests. Comparing results in Table 2, it can be seen that the lower bound latency is 32% and 38% better than the latency obtained when the priority scheme is disabled for workloads WA and WB, respectively. These results will be used to analyze the effectiveness of our scheme.

Table 3 contains the description of the configurations used in our experiments. We use the labels presented in this table to identify the configurations used in any particular experiment. Results for experiments with configurations 1, 2 and 3 show the behavior of our scheme as we increase the maximum number of processes running at higher priority for a fixed maximum number of processes running at lower priority. Experiments with configurations 3, 5 and 6 show the behavior as the number of lower priority processes is increased for a fixed number of higher priority processes. Configuration 4 is just an extreme case when the priority value assigned to type B processes is 0.

Table 4 shows the results obtained for workload WA with both policies: work conserving and non-work conserving. Clearly, the non-work conserving scheme is more effective than the work conserving. The best result obtained when using the work conserving policy was a 18% improvement at a cost of a 115% degradation on the average latency for type B requests. With a non-work conserving policy, the best improvement was 24%; however, the degradation for type B requests was much worse (over 200%).

The difference between the performance of requests of types A and B may seem excessive at a first glance. However, the Webstone-Apache interaction represents a closed system — as soon as a Webstone client gets

Configuration for Workload WA	Average Latency for A (ms)	Average Latency for B (ms)
a) Only requests of type A	283.13	——
b) Requests of types A and B	419.12	419.11

Configuration for Workload WB	Average Latency for A (ms)	Average Latency for B (ms)
a) Only requests of type A	334.71	——
b) Requests of types A and B	544.66	546.49

Table 2: Average Latency for requests of type A and B for both workloads with no differentiated Quality of Service policy (no-policy). a) 15 clients request only files of type A. b) 30 clients request files of both types (15 for each type).

	Type A requests		Type B requests	
Configuration	**Priority Value**	**Max number of processes**	**Priority Value**	**Max number of processes**
Config 1	5	5	1	1
Config 2	5	10	1	1
Config 3	5	15	1	1
Config 4	5	15	0	1
Config 5	5	15	1	2
Config 6	5	15	1	5

Table 3: Description of the configurations used in the experiments.

the response of a previous request, it immediately issues the next request. Since requests of type A have higher priority, a WebStone client that issues a type B request will be pushed back in the processing queue. The overall result is that requests of type A finish much earlier than requests of type B. Though the throughput for type A requests is greater than for type B requests, the sum of them remains the same for all experiments. Table 5 shows the throughput for experiments with configurations 1 and 3 for both work and non-work conserving policies. The throughput for workload WA when the priority scheme is disabled is 27.6 connections/sec and 25.7 connections/sec for type A and type B requests respectively. As can be seen from Table 5, the total throughput remains the same, around 52 connections per second.

For work conserving policy, as the maximum number of processes running with priority A increases, the improvement for requests of type A actually decreases. This indicates that limiting the total number of processes that can concurrently run (including those running with higher priority) reduces contention for resources and improve performance. For non-work conserving policy, the results obtained for workload WA are not very sensitive to the number of processes running with priority A.

Table 6 shows the results obtained for workload WB for both policies. The non-work conserving policy again performs better. The best result was obtained for configuration 4, when the priority value assigned to type B processes is equal to 0. This means that these processes will run only when no type A process is waiting for assuming the CPU. As observed for workload WA, an increase in the number of processes running on priority A actually reduce the performance under the work conserving policy, but not under the non-work conserving policy. Also, comparing results for configurations 3, 5 and 6, we can see that as we increase the maximum number of processes running on priority B, the performance gain for A drops significantly. For a maximum of 5 B requests, the performance gain is lower than 7%. This shows that in order to improve performance for high priority requests, it is necessary to severely restrict the number of processes allowed to run in lower priorities. An increase in this limit from 1 to 2 causes the average latency of type A requests to increase by more than 15% (lines six and seven in table 6).

In order to understand the effect of Linux's timer resolution on the performance, we run the experiments for non-work conserving policy under

Configuration	Work Conserving		Non-Work Conserving	
	Latency for A (ms)	Latency for B (ms)	Latency for A (ms)	Latency for B (ms)
Config 1	344.09 (-18%)	899.48 (+115%)	319.76 (-24%)	1290.96 (+208%)
Config 2	358.54 (-15%)	850.84 (+103%)	319.41 (-24%)	1304.28 (+211%)
Config 3	392.14 (-6%)	717.07 (+71%)	322.25 (-23%)	1501.78 (+258%)

Table 4: Kernel-level approach: average Latency for workload WA. The changes compared to the no-policy results is shown in the parenthesis.

Configuration	Work Conserving		Non-Work Conserving	
	Connection rate for A	Connection rate for B	Connection rate for A	Connection rate for B
Config 1	36.16	15.54	39.90	12.17
Config 3	32.38	20.36	41.44	10.68

Table 5: Kernel-level approach: throughput (connections per second) for workload WA.

$SLEEPING\_PRIORITY = -1$. The process scheduler in Linux is changed to to never choose a process if its priority is equal to -1, even if there is no other process in the ready queue. Thus, a priority equal to -1 means that the process is immediately blocked until its priority is assigned to a non negative value. We set SLEEPING_PRIORITY to -1 under non-work conserving policy where there is no preemption. Table 7 shows the results obtained for this new configuration. As can be seen, effectively blocking a process makes a significant difference in performance. This is basically due to the fact that requests to very small files can be handled very fast. The timing resolution in Linux (10 milliseconds), which affects the time a process with priority 0 can run before being preempted, is actually very long for a web server operation. Even with a priority equal to 0, a process can do some work and increases the contention at the server that, ultimately, affects other processes performance and prevents our scheme from reaching better results.

During the experiments, we observe that the network interface is a bottleneck. We then instrument the kernel to sample the average length of the network interface queue, where IP packets are inserted to be transmitted. The average queue length is around 40 for workload WA and 65 for workload WB.

Thus, in an attempt to improve our results, we extend the process priority to a packet priority scheme. Basically, each outgoing IP packet is assigned the priority of the process that sends it. The priorities are used to sort the network interface queue. Unfortunately, we fail to obtain good results with this scheme. Table 8 show the system performance for workload WB and configuration 3. Apparently the overhead of the sorting contributes to a slowdown in performance. We also suspect that interactions with the network interface card prevent the scheme from being effective. Thus, prioritization at the network interface level needs further investigation.

6 Limitations of the Study

The study is only a preliminary step in investigating differentiated Quality-of-Service (QoS) mechanisms in Web servers. It has not addressed issues such as kernel implementations that circumvent QoS schemes, coordination among scheduling of different resources, and workloads that are more representative of real-life Web server traffic.

Implementation features in traditional kernel networking subsystem can easily defeat any QoS scheme. For example, [14] and [9] point out several problems in traditional kernel TCP/IP implementation that render any user-level QoS scheme a failure under heavy-load. Only a new networking implementation scheme such as the Lazy Receiver Processing [9] can solve the problems. Lazy Receiver Processing would also work well with our priority-based approaches. Combinding the two is part of our future work.

For servers with small main memory, the kernel's file and disk I/O subsystems also affect QoS schemes. Most kernels schedule disk I/O requests regardless of the priority of each request. Thus, unless the Web page is in file buffer cache, the latency for servicing a Web request is subject to the current load on the

	Work Conserving		Non-Work Conserving	
Configuration	**Latency for A (ms)**	**Latency for B (ms)**	**Latency for A (ms)**	**Latency for B (ms)**
Config 1	476.12 (-13%)	724.98 (+33%)	462.72 (-15%)	834.47 (+53%)
Config 2	502.57 (-8%)	651.01 (+19%)	446.63 (-18%)	843.28 (+54%)
Config 3	516.36 (-5%)	593.81 (+9%)	447.94 (-18%)	821.19 (+50%)
Config 4	461.58 (-15%)	1173.68 (+115%)	415.04 (-24%)	1203.65 (+120%)
Config 5	527.57 (-3%)	604.94 (+11%)	480.03 (-12%)	821.32 (+50%)
Config 6	536.21 (-2%)	609.18 (+12%)	507.85 (-7%)	654.88 (+20%)

Table 6: Kernel-level approach: average Latency for workload WB. The changes compared to the no-policy results is shown in the parenthesis.

Config	Latency for A (ms)	Latency for B (ms)	Config	Latency for A (ms)	Latency for B (ms)
Config 1	407.36 (-25%)	1710.82 (+213%)	**Config 4**	404.88 (-26%)	1988.5 (+264%)
Config 2	406.58 (-25%)	1699.93 (+211%)	**Config 5**	437.59 (-20%)	1243.05 (+127%)
Config 3	405.97 (-25%)	1566.92 (+188%)	**Config 6**	518.1 (-5%)	697.37 (+28%)

Table 7: Kernel-level approach: average Latency for workload WB using non-work conserving policy and SLEEPING_PRIORITY equal to -1. The changes compared to the no-policy results is shown in the parenthesis.

disk. For truely differentiated quality-of-service, the kernel needs to change its replacement policy for the file buffer cache and its disk I/O scheduling policy to favor high-priority Web pages.

In general, adjustments of scheduling at all three kernel compoenents (CPU scheduler, buffer cache and disk I/O scheduler, and the networking subsystem) are necessary to provide differentiated quality-of-service under high load. When servicing a Web request, the server process must read the request packet (the networking subsystem), parse the request (CPU scheduler), read the file (buffer cache and disk I/O scheduler), and send the file (the networking subsystem). Our approaches only indirectly affect the scheduling at the three subsystems through the use of process priority and process blocking. Only schemes that directly control the resource scheduling at the three subsystems can provide robust quality-of-service. Investigation of such schemes is out of the scope of this paper. However, we do note that [6] and [17] are recent attempts in this area.

Though we have only studied process-per-request Web servers, the principle of providing differentiated QoS through restricting the number of concurrent low-priority requests applies to other types of Web servers. For example, thread-per-request Web servers can adjust their thread scheduling similar to our user-level approach. Event-driven Web servers such as Squid allow more control at user-level since each step in servicing a request is explicitly scheduled and controlled by the user-level software. We choose to study process-per-request Web servers because Apache uses this architecture and Apache is the most popular Web server software in use.

Finally, Webstone is a simplistic benchmark and cannot drive the Web server to overload [3]. We need to perform tests under other kinds of bursty loads with various mix of high-priority and low-priority requests. This is part of our future work.

7 Conclusions

We have investigated approaches to provide differentiated quality of service by assigning priorities to requests based on the requested documents. We implement the priority-based scheduling at both user and kernel levels. We find that restricting the number of processes that are allowed to run concurrently is a simple and effective strategy in obtaining differentiated performance. Using a closed-queue experiment with WebStone, we are able to observe up to 26% improvement in performance for the higher priority level, with an accompanying 504% fall in performance for B, for the user-level approach with a light workload (WA). For the kernel-level approach, the best improvements are achieved when using a non-work conserving policy. For the light workload (WA) we

Policy	Latency for A (ms)	Latency for B (ms)
Work Conserving	555.74 (-2%)	905.644 (65.72%)
Non-work Conserving (SLEEPING_PRIORITY = 0)	443.317 (19%)	1204.65 (120%)
Non-work Conserving (SLEEPING_PRIORITY = -1)	424.091 (22%)	1440.45 (165%)

Table 8: Kernel-level approach: average Latency for workload WB using configuration 3 and sorting in the network interface queue. The improvement/degradation compared to the no-policy results is shown in the parenthesis.

observe up to 24% of improvement at the cost of a 208% of slowdown for lower priority requests. For the more heavy-tailed workload (WB), the best improvement was 26% at the cost of a slowdown around 260%.

Based on these results, we believe that it is simple and easy to extend a Web server implementation to incorporate differentiated quality-of-service features. Though the user-level approach is the most portable one, tighter integration with the kernel's process scheduling lead to a more robust system that handles diverse workloads.

In choosing between work-conserving and non work-conserving policies, we observe that work conserving policies lose any notion of restriction when the thresholds are large. Thus, we expect that if we have multiple levels of priority, we are likely to encounter overflows of lower priority tasks into higher priority slots, and lose differentiated performance. Thus, it might be better to choose a non-work conserving policy for multiple levels of priority. We are planning to run some experiments with multiple levels of priority to check if our conjecture is true.

Linux's timing granularity affects the performance of the work-conserving policy. In the kernel-level approach, the work-conserving policy is not very effective at preventing the execution of type B processes from interfering with the execution of type A processes. This is because the timing granularity in Linux, which affects the time a low priority process can run before being preempted, is actually very long for a web server operation. Thus, even if a type A process is ready to run, it may not be able to preempt the type B process immediately and gain the CPU. In other operating systems, such as Solaris, where the timer interrupt rate is higher, the work-conserving scheme might work better.

Our results on sorting IP packets in the network interface queue appear preliminary. The results were not consistent and more research is needed on the effect of such schemes.

References

[1] Anderson, E., Patterson, D., Brewer E., The Magicrouter, an Application of Fast Packet Interposing, *Second Symposium on Operating Systems Design and Implementation*, May, 1996.

[2] Arlitt, M. and Williamson, C., Web Server Workload Characterization, *Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[3] Gaurav Banga and Peter Druschel, Measuring the Capacity of a Web Server, *Proceedings of 1997 USENIX Symposium on Internet Technology and Systems*, Dec, 1997.

[4] Banatre, M., Issamy, V., Leleu F. and Charpiot B., Providing Quality of Service over the Web: A Newspaper-based Approach, *Proceedings of the Sixth International World Wide Web Conference*, California, April, 1997.

[5] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R. and Verworner, D., *Linux Kernel Internals*, Addison-Wesley, 1996.

[6] John Bruno, Eran Gabber, Banu Ozden and Abraham Silberschatz, The Eclipse Operating System: Providing Quality of Service via Reservation Domains, *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[7] Crovella, M., and Bestavros, A., Self-similarity in World Wide Web Traffic: Evidence and Possible Causes, *Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[8] Damani, O., Chung, P., Huang, Y., Kintala, C., Wang, Y., ONE-IP: Techniques for Hosting a Service on a Cluster of Machines, *6th International World Wide Web Conference*, April 1997.

[9] Peter Druschel and Gaurav Banga, Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems, *Second Symposium on Operating Systems Design and Implementation*, May, 1996.

[10] Internet Domain Survey, Network Wizards, January,
1997. URL: http://www.nw.cm/zone/summary-reports/report-9701.doc.

[11] Fox, A., and Brewer, E., Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation, *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May, 1996.

[12] Mogul, J., Network Behavior of a Busy Web Server and its Clients, Research Report 95/5, DEC Western Research Laboratory, October 1995

[13] Mogul, J., Operating System Support for Busy Internet Servers, *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[14] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *Proceedings of the 1996 Usenix Technical Conference*, pages 99-111, 1996.

[15] Robinnson, D. and the Apache Group, APACHE - An HTTP Server, Reference Manual, 1995. URL: http://www.apache.org.

[16] Trent, G. & Sake, M. WebSTONE: The First Generation in HTTP Server Benchmarking, February 1995. URL: http://www.sgi.com/Products/WebFORCE/WebStone.

[17] Carl A. Waldspurger and William E. Weihl, Lottery Scheduling: Flexible Proportional-Share Resource Management, *Proceedings of First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1-12, November 1994.