Measurement, Analysis and Performance Improvement of the Apache Web Server *†

Yiming Hu†, Ashwini Nanda‡, and Qing Yang†

† Dept. of Electrical & Computer Engineering
University of Rhode Island
Kingston, RI 02881
{hu,qyang}@ele.uri.edu

‡ IBM T.J. Watson Research Center
P.O.Box 218
Yorktown Heights, NY 10598
ashwini@watson.ibm.com

Abstract

Performance of Web servers is critical to the success of many corporations and organizations. However, very few results have been published that quantitatively study the server behavior and identify the performance bottlenecks. In this paper we measure and analyze the behavior of the popular Apache Web server on a uniprocessor system and a 4-CPU SMP (Symmetric Multi-Processor) system running the IBM AIX operating system. Using the AIX built-in tracing facility and a trace-analysis tool, we obtained detailed information on OS kernel events and overall system activities while running Apache driven by the SPECweb96 and the WebStone benchmarks. We found that on average, Apache spends about 20–25% of the total CPU time on user code, 35–50% on kernel system calls and 25–40% on interrupt handling. For systems with small RAM sizes, the Web server performance is limited by the disk bandwidth. For systems with reasonably large RAM sizes, the TCP/IP stack and the network interrupt handler are the major performance bottlenecks. We notice that Apache shows similar behavior on both the uniprocessor and the SMP systems.

After quantitatively identifying the performance bottlenecks, we proposed 7 techniques to improve the performance of Apache. We implemented all but one of these techniques. Our experimental results show that these techniques improve the throughput of Apache by 61%. These techniques are general purpose and can be applied to other Web servers as well. Finally, our results suggest that operating system support for directly sending data from the file system cache to the TCP/IP network can further improve the Web server performance dramatically.

*To be presented in *the 18th IEEE International Performance, Computing and Communications Conference*, Phoenix/Scottsdale, Arizona, February 1999.

†This work is supported in part by NSF under Grants MIP-9505601 and MIP-9714370, and IBM. Part of this work was performed at IBM T. J. Watson Research Center.

1 Introduction

With the explosive growth of the World Wide Web (WWW), more and more corporations and organizations are depending on high performance Web servers for the success of their business. The high demand of Web requests often stresses or even saturates systems that have very large capabilities. For example, during the first chess match between IBM "Deep Blue" SP2 supercomputer and world Chess champion Gary Kasparov, IBM Web site registered over 5 million hits during the first game and more requests for the following games. IBM had to use 9 SP2 nodes to act as Web servers to handle the heavy Web traffic. According to CNN, NASA Pathfinder Web site topped 100 million hits for the first 3 days after the Pathfinder spaceship landed on Mars. NASA had to set up 20 mirror sites around the world to keep up with the traffic demand. During the election night in November 1996, CNN's Web site recorded 50 million hits.

There are three ways for a Web site to handle high traffic, namely *replication(mirroring), distributed caching, and improving server performance.* Replication is simply distributing the same Web server information to multiple machines that are either a cluster [1], or distributed in different locations. Since any one of the machines can serve requests independently, the load of each individual server is reduced. Distributed caching includes client-side caching [2], proxy caching [3, 4, 5, 6] or dedicated cache servers [7, 8, 9] . These approaches transparently cache remote documents on local storages or a cache machine that is close to the clients, thereby reducing the traffic seen by the original server. Finally, improving server performance includes using more powerful hardware such as a SMP (Symmetric Multi-Processor) system, using better Web server software techniques such as pre-forking process pools [10], as well as using high-bandwidth network connections.

Considerable effort has been invested in studying replication and distributed caching. Many interesting and effective approaches have been proposed and implemented. On the other hand, less attention has been paid to improve the Web server performance. There are very few published results addressing the issue of quantitatively characterizing the behavior of Web servers and improving their performance. Improving Web server performance is important, however. Even with replication and caching, a significant number of requests may still hit the original Web server for a busy Web site. In addition, more and more Web pages are dynamically generated, which can not be cached and must be fetched from the original servers.

To improve Web server performance, it is important to quantitatively identify the performance bottlenecks. In this paper, we measure and analyze the behavior of the Apache Web server running on several different hardware platforms. We show that Apache spends more than 75-80% of its CPU time on OS kernels. We will also give detailed measurement results on where the kernel time is spent. Based on our performance analyses, we present 7 techniques to improve the performance of the Apache server. Together, these techniques increase the throughput of Apache by more than 61% under the SPECweb96 workload.

Expected Throughput (Ops)	Total File Set Size (MB)
1	22
2	31
5	49
10	69
20	98
50	154
100	218
200	309
500	488
1000	690

Table 1: Throughput vs. File Set Size

Classes	File Sizes	Access Frequencies
Class 0	0 – 1KB	35%
Class 1	1KB – 10KB	50%
Class 2	10KB – 100KB	14%
Class 3	100KB – 1MB	1%

Table 2: File Sizes per Class and Frequency of Access

2 Apache, SPECweb96 and WebStone

2.1 The Apache Web Server

Apache [11] is a freely available, UNIX-based Web server developed by a team of volunteers. It was originally based on code and ideas found in NCSA httpd 1.3, but has been completely rewritten since then. We choose Apache as the base of our experimental system for the following three reasons. First, Apache is the most popular Web server running today, accounting for more than 48% of all Web domains on the Internet [11]. Second, Apache is a fully featured, high performance Web server, superior than many other UNIX-based Web servers in terms of functionality, efficiency and speed. Third, the source code of Apache is available, enabling us to make changes to the code to improve its performance. We used Apache 1.2.0 during the experiment.

Apache is a multi-processed program. When Apache starts, the main process forks into several child processes that are responsible for handling incoming HTTP requests. Each child process listens to one or several specific TCP ports (normally port 80). When a HTTP request comes in, one child gets the request. This child process reads the request, parses the URL (Uniform Resource Locator), finds the file name corresponding to the URL, checks the file states, performs security checking, opens the file, reads its content, and finally sends the content to the client. The child process then logs the request information to a log file, and listens to the next request.

2.2 The SPECweb96 Benchmark

SPECweb96 was developed by the Standard Performance Evaluation Corporation (SPEC). It is the first standardized benchmark for measuring the performance of Web servers.

SPECweb96 consists of two parts: a *file set generator* and a *workload generator*. Before testing a Web server, the file set generator must be run in the server machine to populate a test file set consisting of many files of different sizes. The workload generator then runs on one or several computers connected to the Web server machine being tested via a TCP/IP network. It simulates Web clients by sending HTTP "GET" commands to the Web server, requesting for files in the test file set. The workload generator measures the response time of the Web server, and increases the request rate until the server can not handle them. The maximum HTTP request rate that the Web server can handle is the SPECweb96 value.

An important feature of SPECweb96 is that the total size of the file set scales with the expected throughput of the server. This is because SPEC believes that a higher-end server should not only provide faster services but also serve more files (Web pages) than a smaller server can. Table 1 shows the relationship between the expected throughput and the total file set sizes defined by SPECweb96.

Many studies indicate that small files are accessed more frequently than large files in most real world Web servers [12, 13, 14]. Similarly, in SPECweb96, the data files are classified by their sizes into 4 classes, as shown in Table 2. Smaller classes are accessed more frequently by the workload generator than larger classes are. The file sizes and access frequencies are based upon analysis of logs from several popular servers, including NCSA's site, the home pages for Hewlett-Packard and HAL Computer.

In the SPECweb96 file set, Class 3 files account for 90% of the total file set size. However their chances of being accessed are only about 1%. The remaining files account for only 10% of the total file set size, yet receiving 99% of total accesses. This leads to an important conclusion. That is, a relatively small amount of RAM can be used to cache the file set and eliminate most disk accesses. For example, a 4-way SMP system can typically achieve a SPECweb96 number of 2000, corresponding to a file set of 1 GB. Only 100 MB of RAM is required to cache the file set to achieve a document hit ratio of 99%, or a byte hit ratio of about 60%. In fact, most high performance server systems nowadays have memory sizes of hundreds megabytes or even several gigabytes, as demonstrated by SPECweb96 reports. These systems effectively cache almost all files in RAM. As a result, for such systems, disk activities have little effect on the overall server performance.

2.3 The WebStone Benchmark

SPECweb96 is a standardized benchmark that generates relatively realistic Web workloads for evaluating the overall system performance. It mixes 4 different classes of files in a way close to the access patterns observed in real world Web servers. However, researchers often need to change the characteristics of the workload, such as the request sizes, to identify the server performance bottleneck under a specific condition. SPECweb96 does not allow users to change the workload. Therefore, when we need to study the behavior of Apache under different request sizes, we will use the WebStone benchmark developed by SGI, which gives users almost complete control over the workload characteristics, including the request sizes and mixtures.

One problem of WebStone is that its default file generator populates the server with only 32 files. If we use these files as the server file set, the server will transfer a very small number of files over and over again. This may generate misleading results, since the small number of files will always stay in the file cache and the CPU cache. Fortunately, WebStone allows using any data files. We choose the same SPECweb96 data files, which is large enough (several hundreds megabytes), as the Server data file. We setup WebStone to selectively request the SPECweb96 files on a specific size range.

3 Experimental Environments

We studied the behavior of the Apache server on both a uniprocessor system and a SMP multiprocessor system. The uniprocessor experimental system consists of a IBM RS/6000 system as the server and two Pentium PC systems as clients, as shown in Table 3. Each client machine has a dedicated 100 Mbps Ethernet connection to the server. The SMP experimental system consists of two state-of-the-art IBM 4-Way SMP machines with 2 GB of RAM each, one as the server and the other as a client, connected through two dedicated 100 Mbps Ethernets.

Server Machine (1 RS/6000)	Model	43P-140
	CPU Type	200 MHz PPC 604e
	RAM Size	128 MB
	Disk Space	2 x 2 GB
	OS	AIX 4.2.1
Client Machines (2 Pentium PCs)	Model	Pentium PC
	CPU Type	133 MHz Pentium
	RAM Size	32 MB
	Disk Space	2 GB
	OS	Linux 2.0.30
Network	2 x 100 Mbps Ethernet	

Table 3: Uniprocessor Test System Configuration

We use the AIX built-in trace facility to capture the system activities while Apache is running. The trace facility records kernel events with an extremely fine granularity of details [15]. The collected trace data is post-processed by the UTLD 1.2 program [16] developed by IBM RISC System/6000 Division. UTLD analyzes the trace file and generates a detailed system utilization report including the CPU utilization, locking time, interrupt handling time, as well as CPU times of individual system calls. For better measurement accuracy, we repeat each measurement 3 times and average the results.

The two server systems used by this study have large RAM sizes. To study the behavior of Web servers under smaller RAM sizes, we run the AIX *rmss* (Reduced-Memory System Simulator) command [17] to simulate systems with different sizes of real memories that are smaller than the actual memory size, without having to extract and replace memory boards.

4 Measurement Results

4.1 Effects of System RAM sizes

In our first experiment, we run Apache on the uniprocessor system with different RAM sizes. Using *rmss*, we set the usable system RAM sizes to 16, 32, 64 and 128 MB, and run the SPECweb96 benchmark to drive the Web server. Figure 1 shows the achieved SPECweb96 throughput (the number of HTTP requests per second) as well as the network bandwidth (the number of bytes transfered per second), both differ dramatically under different system RAM sizes.

Our analysis indicates that the performance differences are mainly caused by the effect of disk caching. For the workload range we considered, SPECweb96 touches about 100-380 MB of file data during a typical test run. When the RAM size is small, such as in the cases of 16 and 32 MB, the system RAM is used up mainly by the program code and data pages. As a result, most files can not be

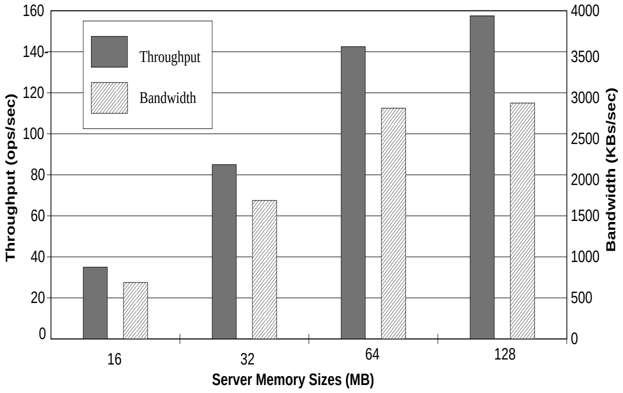

Figure 1: Apache SPECweb96 Performance

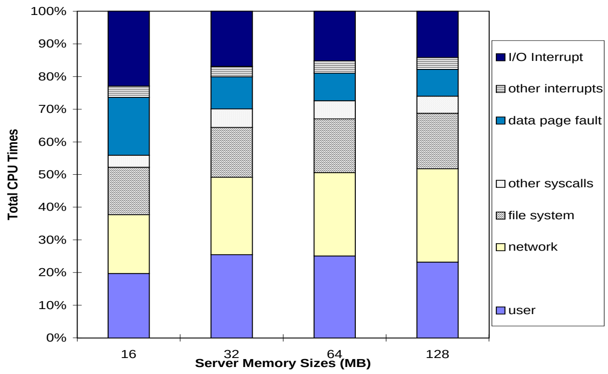

Figure 3: Apache CPU busy-time breakdown

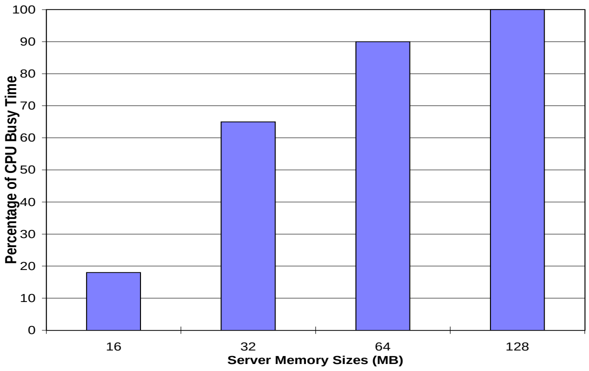

Figure 2: Apache CPU Utilization

cached in the RAM and must be fetched from the disk. In fact, we observed that in such cases the disk is 100% busy most of time, meaning that to get the data for the next Web request, the CPU must wait until all current disk requests in the disk queue finish. In other words, the Web server performance at low system RAM sizes is limited by the disk bandwidth. This can be further demonstrated by Figure 2, which shows the CPU utilization (the proportion of time that the CPU is busy). For the system with 16 MB of RAM, the CPU does useful work for less than 20% of time. For the rest of the time the CPU simply waits for disk I/Os. On the other hand, the CPU is almost 100% busy for the system with 128 MB of RAM.

We also traced the system activities and obtained detailed timing information of the system. Figure 3 shows the CPU activities as percentages of the total CPU busy time. In general, Apache spends only about 20-25% of its CPU time on the user code. On the other hand, it spends almost 35-50% of its CPU time on kernel system calls, which includes the "file system", the "network" and the "other syscalls" portions as shown in Figure 3 [1]. The remaining 25-40% of the CPU time is for handling first level interrupts including I/O interrupts (Ethernet and disk interrupts), page faults and other interrupts(clock interrupts, etc.). The interrupt handlers belong to the bottom half of the OS kernel [18]. Technically they are not parts of the Web server processes. However, they provide necessary services for the Web server and consume a very significant portion of the CPU time.

As pointed out before, the Web server performance is limited by the disk bandwidth for small system RAM sizes. There is perhaps little that one can do to improve the server performance under small RAM sizes, except for using fast disk systems or adjusting the disk caching algorithms. On the contrary, the Web server performance on a system with a large RAM size is mainly limited by its CPU performance, since our profiling data show that the CPU is saturated. If we can identify the performance bottlenecks and remove them to reduce their CPU time requirements, we can improve the server performance. In the following discussions we will concentrate on studying the behavior of Web servers on systems with large RAM sizes. All following results for the uniprocessor system are obtained with 128 MB of RAM. For the SMP system, all results are measured under the default setup of 2 GB of RAM.

Figure 4 gives detailed CPU time breakdown for Apache running on the uniprocessor system with 128 MB of RAM. It clearly shows that the TCP/IP stack and the low level network handling consume the majority of the CPU resource. The code directly involved with TCP/IP (the dark sectors on the bottom, including the *Ethernet Driver*, the *write* [2], *select* and other TCP/IP system calls) takes 29%

[1] We include the Ethernet and disk drivers as the TCP/IP stack and File system activities because the device drivers are closely related to the TCP/IP stack and File system. However the device drivers are actually called by Second Level Interrupt Handlers. Thus strictly speaking, they are bottom-half kernel activities and are not parts of the Web server processes.

[2] A very small percentage of the write time is actually caused by writing the log file, which is a file system activity. Similarly, a small portion of the *read* time (about 10%) is caused by read requests from the network, although we classify *read* as a file system operation.

of the total CPU time. The first level interrupt handlers (the light sectors on right, including *I/O Interrupt, data page fault* and *other interrupts*) also use a significant amount (24%) of CPU time. Among them, the *I/O Interrupt* portion is mainly for handling Ethernet interrupts. Since most disk files are cached in RAM, there are very few disk activities observed. The TCP/IP and the Ethernet interrupt handlers together take 43% of the total CPU time, which is the main performance bottleneck. File system operations (the light sectors on the left, including the *read*, *open*, *stat* and other system calls) use 17% of the CPU time, which is very wasteful because more than 99% of the active data files can be cached in the system RAM. The user code uses about 23% of total CPU time. Finally, the "incinterval" system call that is mainly used by the *alarm* and other timer functions uses 3% of the processing time. All other miscellaneous syscalls, which include signal handling and system calls such as *getpid* and *exit*, take only about 2% of the total CPU time together.

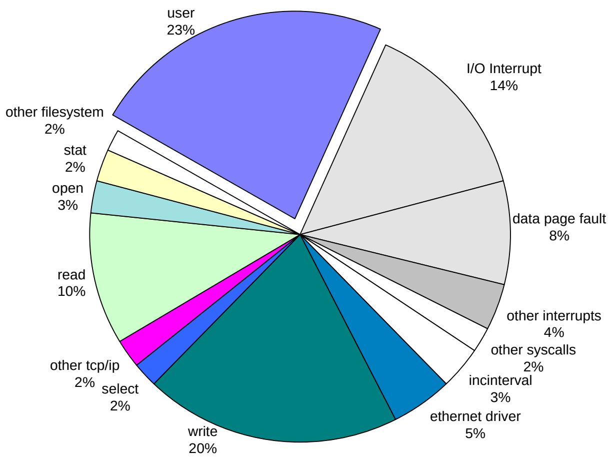

Figure 4: Detailed Apache CPU Time Breakdown (128MB)

We have also run Apache on a 4-CPU SMP system with 2 GB of RAM as a server and SPECweb96 on another SMP system as a client to generate HTTP requests. We found that the behavior of Apache on a SMP system is very similar to that on a uniprocessor one. Interested readers can refer to our technique report [19] for a detailed description of the behavior of Apache on the SMP machine.

4.2 Effects of Request Sizes

SPECweb96 generates Web requests with mixed request sizes. While it gives us a balanced view on the overall system performance and behavior, it is often desirable to study the system behavior under different request sizes. The studies of ours and others [20] indicate that different request sizes stress the system in very different ways.

To study the behavior of Apache under different request sizes, we conducted another 4 groups of tests on the uniprocessor system, using the WebStone benchmark. For the reasons described in Section 2, we use SPECweb96 data files as the test file set for WebStone. The SPECweb96 files are classified according to their sizes with class 0 being the smallest sizes and class 3 the largest sizes. In each group of the tests we let the WebStone request about 100 different files from a specific SPECweb96 file class. As a results, we are able to measure the behavior and identify the problems of Apache under different file sizes.

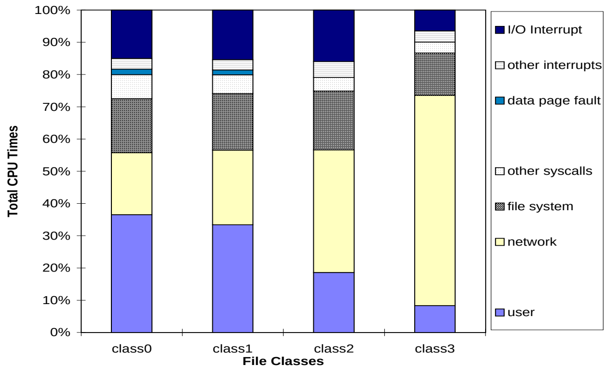

Figure 5: Apache CPU Time for different file sizes

Figure 5 summarizes the results of the Apache behavior under different request sizes. This figure clearly shows that when the request sizes are small (class 0 and class 1 files), user time dominates the total CPU time. The file system and the TCP/IP stack are relatively equally stressed. For large requests (class 2 and class 3 files), Apache spends most of its time on the TCP/IP stack.

The detailed CPU time breakdowns for different file sizes are shown in Figures 6 to 9. It is clear from the figures that, for small request sizes (class 0 and class 1), Apache user code is the main performance bottleneck, accounting for 34-36% of the total CPU time. This is because that there is a fixed amount of overhead, such as parsing the request and logging the request, involved in each request. When the request size is small, the time spent on transferring data from the file system to the network is relatively short, therefore this fixed overhead becomes a dominant portion. The large overhead suggests that there is potentially a large room of improvement on the user code of Apache. First level interrupt processing is another major overhead, consuming 19-20% of the total processing time. However, there is perhaps little one can do about this, except for using a very intelligent device that requires less CPU attention. Finally, the CPU spends almost identical amounts of times on the file system and the TCP/IP stack. There are no obviously dominant system calls, implying that one can not expect a significant performance increase by eliminating one or a few system calls.

The cases for large file sizes are completely different. Because of the large request sizes, the overhead of the user

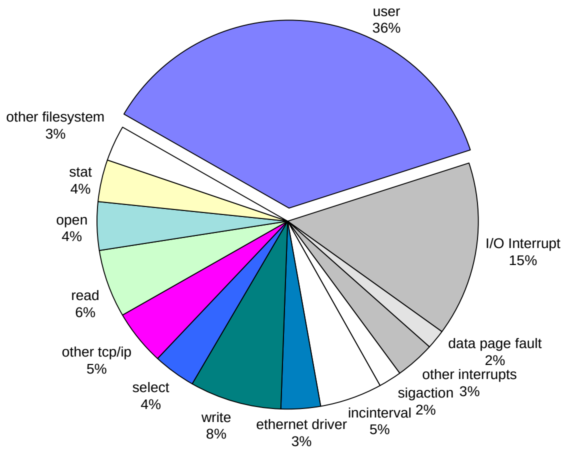

Figure 6: Detailed CPU Time Breakdown for Class 0 files

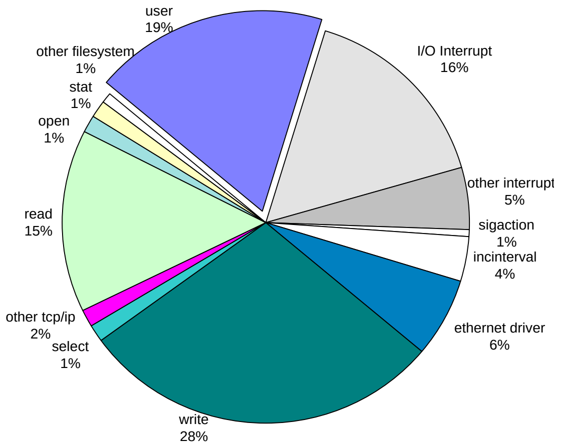

Figure 8: Detailed CPU Time Breakdown for Class 2 files

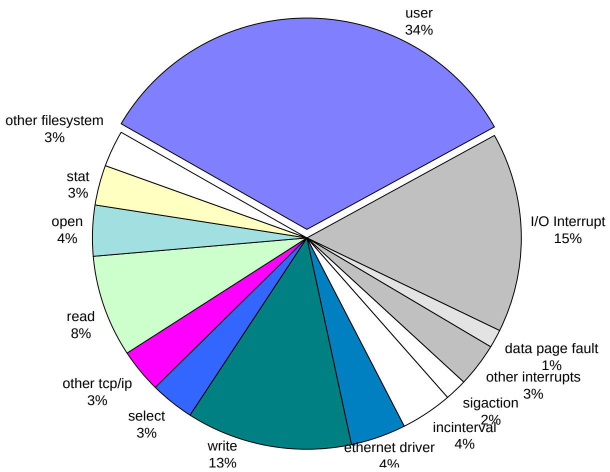

Figure 7: Detailed CPU Time Breakdown for Class 1 files

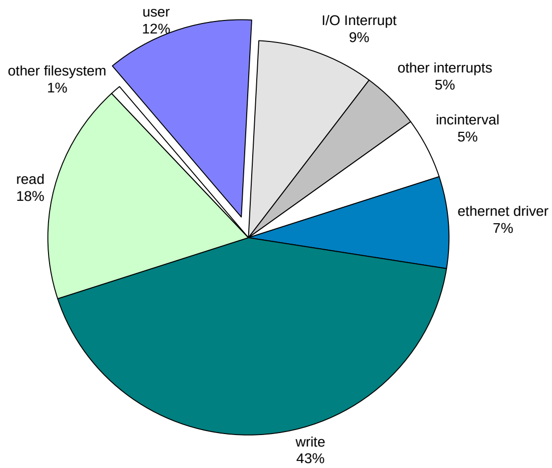

Figure 9: Detailed CPU Time Breakdown for Class 3 files

code becomes a relatively small portion of the total processing time. Rather, the CPU spends most of its time on the *write* system call, which copies data from the user space to the kernel $MBUFs$ (the buffer structures used by the TCP/IP stack). The total time spent on the file system is similar to the cases of small file sizes. However, now the *read* system call, which transfers file data from the file system cache to the user space, dominates the file system operations because of the large file sizes. The overheads caused by other file system activities, such as *open* and *statx*, become negligible.

5 Improving the Apache Performance

We have shown that for the SPECweb96 workload, on a uniprocessor system Apache spends about 23% of the total CPU time on user code, 26% on interrupt handlers, 29% on TCP/IP processing and 17% on file system operations. It is difficult to reduce the overheads caused by TCP/IP processing and interrupt handling unless we modify the OS code or change the hardware. On the other hand, it is possible to greatly reduce the file system overhead and the user code overhead, as will be demonstrated in the remaining of this section.

5.1 Using the *mmap* function

For our system with 128 MB of RAM, almost all frequently accessed file data are cached in the file system cache. However, Figures 6 to 9 clearly indicate that for all 4 different classes of files, Apache still spends a considerable amount (17-19%) of its CPU time on file system activities. The reason to this high overhead is that for almost every HTTP request, Apache has to check the file state using the *stat* system call, *open* the file and *read* the file data from the file system cache into the user buffer, before it can ship the data to the network.

Our first approach is to use the *mmap* function to eliminate the data copying between the file system cache and the user space. When Apache needs to read a file, we let the program open the file, then use the *mmap* function to map the data of the file into the user space. As a result, the server can send the mapped data directly to the network, avoiding the *read* system call altogether. When the entire file content is sent out, the file is unmapped and closed to limit the number of opened files in the system (most OSs pose a limitation on the number of files a process can open).

This approach is easy to implement. However, we found that the effectiveness of this approach is limited for the following two reasons. First, for small file sizes (class 0 and class 1 files), Figures 6 and 7 show that the *read* system call accounts for only 30-50% of the total file system overhead. Other file system calls that can not be eliminated by *mmap*, such as *open* and *close*, also have significant impacts on the server performance. Second, for each mmapped pages accessed by the Web server, the Virtual Memory system generates a page fault, even the file data are already cached in the kernel buffer. Since the file is unmapped and closed after the data are sent out, a sequence of page faults will occur again when the file is re-mapped into the user space next time. Because of the overhead of page faults, reading data from the mmapped area is about 20-100% slower than from the user memory space in most OSs, as reported by McVoy and Staelin [21]. Our own experiments also confirmed their observation. The *lmbench* benchmark reports that our system has a memory bandwidth of 72 MB/sec for reading data from the user space. The memory bandwidth is 59 MB/sec for reading data from mmapped areas, which is about 22% lower than the former. We will further discuss the performance impact of the *mmap* overhead shortly.

5.2 Caching files in the User Space

A better solution is to cache the file data in the Web server user space. A Web server can directly ship the data of cached files to the TCP/IP stack, avoiding all file system calls such as *open*, *read* and *close*. Moreover, as discussed above, reading data cached in the user memory space is faster than from the mmapped area, resulting in a better server performance.

To evaluate the performance benefit of caching in the Web server user space, we conducted the following experiment. we let the WebStone repeatedly retrieve a single 1 MB file from the Apache server. Because of the large file size, the overheads caused by *open* and *close* system calls are negligible. We have already found that in our system, reading data from the user space is about 22% faster than from a mmapped area. Figure 9 shows that when transferring large files, Apache spends most of the time on the *write* function, which reads data from the user space or from the mmapped area and writes the data to network buffers. Therefore we expected that the server bandwidth of Apache caching data in the user space should be noticeably higher than that of Apache using *mmap*. We obtained an average server bandwidth of 87.24 Mb/sec by caching the file in the user space, as compared to 78.30 Mb/sec if we use *mmap* to fetch the file data, giving rise to almost 11% improvement.

Our profiling data also indicates that Apache frequently calls the *stat* system call to get the file states, such as the access permission and create-time information. For small file sizes, the *stat* system call accounts for about 20% of the total file system overhead. It will be beneficial to cache the file state in the Web server user space also. Since the file state information is small, the overhead for caching this information is minimal.

We have implemented a user-level cache for Apache. Because of the space limitation, we will not discuss the implementation details in this paper. Interested readers can refer to our technique report [19] for more information. Basically, the cache stores both the file data and the file states in a user memory region shared by all processes. When the server tries to read the content or state of a file, the cache is searched first. On a cache hit, information stored in the cache is returned immediately. On a cache miss, the cache invokes system calls to obtain data, put the data in the cache, and return them to the server.

5.3 Speeding up Logging

Typically a Web server logs every HTTP request into a log file. The logging information includes the client name, the request time, the requested URL, etc. During our experiments we found that the logging process of Apache is a major overhead. For example, on the uniprocessor system with 128 MB of RAM, the SPECweb96 number of Apache with logging is 158 ops/sec. Simply turning off the logging operation results in a SPECweb96 number of 194 ops/sec, which is a 22% improvement. Since logging information is very useful for many Web sites, it is desirable to reduce the logging overhead of Apache.

5.3.1 Caching DNS results

We found that the majority of overheads of logging comes from looking up the host names of clients. Apache uses the *getnamebyaddr* function to perform the DNS (Domain Name Service) lookup using client IP addresses as inputs. Apache calls the function for every HTTP request being logged. This is very wasteful, since users normally send out a sequence of requests to a Web server for multiple objects in a Web page (the HTML file and many small bitmap files). As a result, multiple name lookup operations are performed for the same IP address. The overhead is especially high if there is no name server running at the local machine. In such a case, the name lookup requests must be sent across the network to a name server, causing long delays and extra network traffic. For small file sizes, the logging delays caused by DNS lookups may dominate the HTTP request response time. While it is possible to disable the host name lookup processes and log only the client IP addresses, most Web administrators would prefer logging host names, because it provides much more information.

We solve the problem by using a simple and effective technique — A DNS cache that caches the host names and IP addresses of clients in the Web server address space. The DNS cache is a small array (about several hundred entries) of records that contain the host names and IP addresses of client machines. The cache entries are indexed by hashing the IP address of machines. When Apache needs to lookup a client's name with an IP address, the DNS cache is checked first. If there is a cache hit, the machine name is returned from the cache. Otherwise the cache calls the *getnamebyaddr* function first to find the machine name, puts the information into the cache, then returns the machine name. Our measurement results show that this simple solution improves the throughput of Apache by more than 14%. This is equivalent to reducing the logging overhead by 63% (14/22), since the logging overhead reduces the server throughput by 22%.

5.3.2 Caching String Results

We ran Apache with the *gprof* profiling tool and found that another noticeable portion of the logging overhead is caused by logging the request time and status code. For each request, Apache has to convert the current system time into a human-readable ASCII string and then write the string into the log file. Similarly, it has to convert the request status code — success or fail, for example — into an ASCII string. Both string conversion operations are expensive.

However, many of the string operations are redundant. For example, the logging time resolution is only 1 second. As a result, in any second, Apache calls the time conversion routines hundreds times, only to convert the same time into the same string again and again. Similarly, the majority (80-90%) of status codes during a normal operation period should be "successful" [12]. It is wasteful to convert the same "successful" status code into the same string over and over again.

The overhead can be significantly reduced by caching the last string conversion result. For example, we let the time-conversion routine store the resulting string in a static array. When Apache calls the conversion routine again, the routine returns the result in the static array if the current time argument is the same as the last call, thus avoiding many redundant string operations. Similar optimizations can be applied to the status code conversion and several other places in the Apache code. This requires changing only several lines of code. The result is a 20% of reduction on the logging overhead, or a throughput improvement of about 4 percent.

5.4 Caching URI Processing Results

We have shown that Apache spends about 20-23% of its CPU time on user code. Using the *gprof* tool, we found that about 60% of the user time are for processing URIs (Uniform Resource Indicators, the parts of URLs after the colon), e.g., parsing, directory checking, security checking, translating the URI to a file name, etc. The URI parsing overhead becomes a major performance bottleneck after the file system overhead is eliminated by the user-level file cache.

The problem here is very similar to the one faced by physically addressed CPU caches. A physical CPU cache can eliminate the overhead of transferring data between the main memory and the CPU on cache hits. However, the CPU needs to translate virtual addresses to physical addresses before accessing the cache. The address translation process is potentially very expensive, involving accessing page tables in the RAM for each memory access. Almost all modern CPUs use TLBs (Translation Look-aside Buffers) that significantly reduce the address translation overhead. A TLB caches the translated physical addresses to speed up address translation. We adopted the idea of TLB into the Web server design. The URI parsing process resembles the process of translating a virtual address (a URI) to a physical address (a file name). This similarity leads us to the design and implementation of a URI cache that greatly speeds up the URI processing of Apache. When Apache finishes parsing and checking a URI and obtains a file name corresponding to the URI, the URI/file-name pair is put into the cache. Later when a client send the same URI to the server, the server can obtain the file name corresponding to the request directly from the cache and send the data out, without going through the time-consuming parsing and checking process again. Note that because Apache supports the concept of "Virtual host" [3], the same URI may result in different file names. Consequently, each virtual host has its own URI cache. Similarly, different URIs may correspond to the same file, although it is easy to handle this situation.

This scheme is especially helpful for dynamically generated Web pages such as directory lists, because it is expensive to generate such pages. Because dynamic pages do not have corresponding physical files, the program assigns a unique "pseudo-file-name" to each page and stores the resulting page data with the pseudo-file-name in the file cache. The URI/pseudo-file-name pair is also cached in the URI cache to speed up accesses.

Another possible solution is to use the idea of virtual address caches. For example, the file cache can be indexed by URIs (the virtual addresses) instead of by file names (the physical addresses). When a client requests a URI, the file cache is searched using the URI as the index, and the file content is returned on a cache hit. This solution is

[3] "Virtual hosts" refers to the ability of a single Web server acting as multiple "virtual" Web servers, each of them has its own name and IP address.

faster than the decoupled URI-Cache/File-Cache solution presented above. However, after some experiments we rejected this idea. The reason is that this scheme faces the same problems of *synonyms*(*aliases*) and *homonyms* of virtual caches. Synonyms occur when several URIs correspond to the same file name. A virtual cache will cache the same file multiple times for each URI, wasting memory space and may cause a consistency problem if the Web server supports the HTTP "PUT" (modification) operation. Homonyms occur when a URI corresponds to several different file names, such as in the case of multiple virtual hosts. Solving the problems of synonyms and homonyms requires relatively complex software, therefore we implemented the simpler and more flexible decoupled approach.

Apache supports "Content Negotiation". For the same URI, Apache may return different documents, say a French version or an English version, upon the request of clients. Currently we have not found an efficient way to handle content negotiations in the URI cache. Therefore, we do not cache the URI/file-name pair if it is a result of a content negotiation. This will not cause a major performance problem, since not many documents nowadays use content negotiations.

6 Results of the Enhancement Techniques

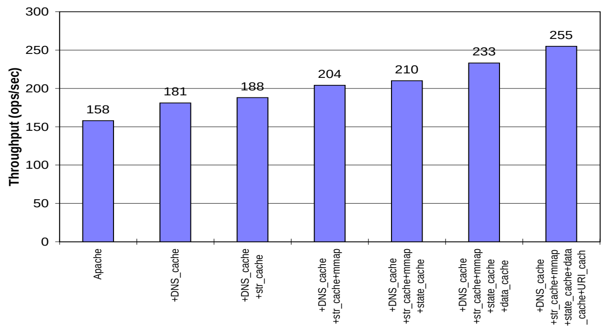

Figure 10: SPECweb96 Results of the Enhanced Apache

Figure 10 compares the performance of Apache with various performance-enhancement techniques we proposed and implemented. The bar on the far left is the performance of the original Apache. We can see that the DNS cache improves the server throughput by more than 14%. Caching the string conversion brings in another 4% performance increase. We then use the *mmap* function to speed up both small files and large files accesses. This results in a 9% throughput increase. However *mmap* does not eliminate other file system calls such as *open* and *close*, therefore the server still spends a lot of time on file system operations. Next we add the cache of the file states, which eliminates all *stat* calls and results in a 3% improvement. We then add the user-level file cache for small files, and use *mmap* only to speed up accesses for uncached large files. The user-level file cache removes almost all file system calls, improving the performance by about 11%. Finally, we implement the URI cache and obtain another 10% speed up. Together, the 6 techniques we proposed and implemented boost the performance of Apache by more than 61%, as shown by the right most bar in Figure 10.

While it is possible to further optimize the user code of Apache, two activities, namely the Ethernet I/O interrupts and the *write* system calls, greatly limit the server performance. Although it is difficult to reduce the interrupt overhead, it is possible to greatly reduce the overhead of *write* system calls for the following reasons. The overhead of a *write* is mainly caused by data copying. Even with *mmap* and our user-level file cache, the *write* system call still has to copy data to the kernel $MBUF$, either from the kernel file cache in the case of using *mmap*, or from the user file cache. The data copying causes significant overhead, especially in the cases of multiprocessor systems because of the cache-coherence protocols. Moreover, the copying process flushes the CPU data cache, increasing the cache miss ratios.

We believe that future operating systems should provide support for directly sending data in the kernel buffer to the network without copying. For example, the OS can provide a system service called "read_file_into_mbuf", which reads the file data from the disk and caches them in the kernel in an $MBUF$ chain. Later an application can issue a "send_file_in_mbuf" system call, which passes the $MBUF$ chain directly to the TCP/IP stack, without the need of data copying. This will greatly improve the performance of Web servers as well as other network applications such as file servers. It is also possible to unify the above two calls into a single "send_file" system call, which ships the specified file from the disk (if the file is not cached) to the network directly [4]. This technique is especially important for large files, because Figures 8 and 9 show that a Web server spends most of its time on the *write* system call for large file sizes. We expect that the technique will at least double or even triple the server performance for large files which are becoming more and more important because of the increasing use of audio and video files on the Web pages. It will also significantly reduce the server overhead for small files.

7 Related Work

McGraph [22] measured the throughputs and response times of several Web servers on 4 different hardware platforms. He found that delivering large files is dominated by the network transfer time, regardless of the server software or platforms.

[4] Microsoft already has such an API called $TransmitFile$ in the Windows NT system.

Almeida et al. [20] and Yates et al. [23] have recently presented an interesting study on measuring the behavior of the Apache Web server on a PC running Linux [20]. Using the Webmonitor that they developed and the kernel profiling facility in Linux, they have measured and analyzed the server performance on top of Linux. Our research differs from theirs in the following important aspects. First, the AIX tracing facility and the UTLD program provide us with much more detailed system activity statistics than Linux can. Second, Almeida and Yates et al. used a low-end platform, which is a 75 MHz Pentium PC with 16 MB RAM. We study the Web server behavior on a range of hardware systems, such as a fast 200 MHz PowerPC uniprocessor system with 128 MB of RAM, and a very powerful 4-Way SMP system with 2 GB of RAM. As indicated previously, different memory sizes present quite different performance behaviors of the Apache server. Third, they used a 10 Mbps Ethernet as the connection between the clients and the server. Our studies, as well as the SPECweb96 documents, indicated that the 10 Mbps Ethernet is saturated by a few clients, leaving the CPU idle most of time and causing notable measurement errors. Fourth, instead of only using WebStone, we also use SPECweb96, which is a standardized benchmark and generates more realistic workload. Finally, we have also proposed and implemented several performance enhancement techniques that improve the performance of Apache significantly.

In [24] Markatos proposed caching Web documents inside the address space of Web servers, which is similar to our user-level caching approach. He referred to this idea as "Main Memory Web Caching". He used server traces from several Web sites to conduct trace-driven simulations. He showed that even a small amount of main memory (512 Kbytes) can hold more than 60% of the documents requested. We independently proposed the similar idea of caching file data and actually *implemented* the cache for the Apache Web server. We also use the *mmap* function to speed up accesses to large files. Moreover, in addition to the file content, we cache file states also, since our profiling data show that Apache frequently inquires the states of files. Our measurements under the SPECweb96 workload demonstrated that our cache design and implementation is successful. Finally, we use the OS virtual memory system to implement the cache LRU algorithm. This approach greatly simplifies our design and interacts well with other parts of the system.

Several new Web servers, such as *Zeus*, also use the *mmap* function for fast file accesses. However, because of the lack of documentation, we do not know their implementation details. Hu et al. have also independently proposed the ideas of using file caching and *mmap* in [25].

8 Conclusions

In this paper, we present measurement results and performance analyses of the behavior of the Apache Web server on a uniprocessor system and a 4-CPU SMP system running the IBM AIX operating system. Using the built-in tracing facility and a trace-analysis tool, we obtained detailed information on OS kernel events and overall system activities while running Apache driven by the SPECweb96 and the WebStone benchmarks. We found that, on average, Apache spends about 20-25% of the total CPU time on user code, 35-50% on kernel system calls and 25-40% on interrupt handling. For systems with small RAM sizes, the Web server performance is limited by the disk bandwidth. For systems with reasonably large RAM sizes, the TCP/IP stack and the network interrupt handler are the major performance bottlenecks. We also believe that using a thread-based Web server architecture instead of a process-based approach may not always improve the server performance noticeably. We notice that Apache shows similar behavior on both the uniprocessor and the SMP systems.

No	Technique	Implement. Complexity	Effect-iveness
1	*Mmapping* large files	0	2
2	Caching small files	2	3
3	Caching file states	1	1
4	Caching DNS	0	3
5	Caching string results	0	1
6	Caching URI process results	2	3
7	Unifying FS/Network caches	3	3

Table 4: Web Server Enhancement Techniques. *For complexity, 0 means trivial and 3 means challenging. For effectiveness, 0 means almost useless, while 3 means very effective.*

After quantitatively identifying the performance bottlenecks, we proposed 7 techniques to improve the performance of Apache, which are summarized in Table 4. We have implemented all but the last one techniques listed in the table. Our experimental results show that these techniques are quite effective. Together they improve the throughput of Apache by 61%. These techniques are general purpose and can be applied to other Web servers as well. Finally, our results suggest that operating system support for directly sending data from the file system cache to the TCP/IP network (the last technique listed in Table 4) can further improve the Web server performance.

Acknowledgments

Erich Nahum at IBM Watson read an early draft of this paper and gave many constructive comments and suggestions that greatly improved this paper. We benefited from discussions with Rich Neves at IBM Watson and Pedro Trancoso at University of Illinois. Bulent Abali at Watson helped us on setting up the test network.

References

[1] E. D. Katz, M. Butler, and R. McGrath, "A scalable web server: The NCSA prototype," in *WWW'94 Conference Proceedings*, 1994.

[2] A. Bestavros, R. L. Carter, M. E. Crovella, C. R. Cunha, A. Heddaya, and S. A. Mirdad, "Application-level document caching in the internet," in *Proceedings of the Second Intl. Workshop on Services in Distributed and Networked Environments (SDNE'95)*, 1995.

[3] A. Luotonen and K. Altis, "World-Wide Web proxies," in *WWW'94 Conference Proceedings*, 1994.

[4] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox, "Caching proxies: Limitations and potentials," in *Proceedings of the Fourth International Conference on the WWW*, (Boston, MA), Dec. 1995.

[5] C. Maltzahn, K. J. Richardson, and D. Grunwald, "Performance issues of enterprise level web proxies," in *Proceedings of the 1997 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[6] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *USENIX Symposium on Internet Technologies and Systems (USITS)*, Dec. 1997.

[7] J. Gwertzman and M. Seltzer, "The case for geographical pushcaching," in *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[8] S. Glassman, "A caching relay for the World Wide Web," in *WWW'94 Conference Proceedings*, 1994.

[9] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," in *Proceedings of the 1996 USENIX Technical Conference*, (San Diego, CA), Jan. 1996.

[10] A. Cockcroft, "Watching your Web server." http://www.sun.com/sunworldonline/swol-03-1996/swol-03-perf.html, Mar. 1996.

[11] The Apache Team, "Apache HTTP server project." http://www.apache.org/.

[12] M. F. Arlitt and C. L. Williamson, "Web server workload characetrization: The search for invariants," in *Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[13] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing reference locality in the WWW," in *Proceedings of the 1996 IEEE Conference on Parallel and Distributed Information Systems*, Dec. 1996.

[14] M. E. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic: Evidence and possible causes," in *Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Philadelphia, PA), May 1996.

[15] D. A. Helly, *AIX/6000 Internals and Architecture*. New York, NY 10020: McGraw-Hill, 1996.

[16] IBM RS/6000 Division, "Utld 1.2 user's guide."

[17] IBM, "AIX versions 3.2 and 4 performance tuning guide."

[18] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley, 1989.

[19] Y. Hu, A. Nanda, and Q. Yang, "Measurement, analysis and performance improvement of the apache web server," Tech. Rep. 1097-0001, Department of Electrical & Computer Engineering, University of Rhode Island, Oct. 1997.

[20] J. M. Almeida, V. Almeida, and D. J. Yates, "Measuring the bahavior of a world-wide web server," in *Seventh Conference on High Performance Networking (HPN)*, (White Plains, NY), pp. 57–72, Apr. 1997.

[21] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 USENIX Conference*, 1996.

[22] R. E. McGrath, "Performance of several web server platforms." http://www.ncsa.uiuc.edu/InformationServers/ Performance/Platforms/report.html.

[23] D. J. Yates, V. Almeida, and J. M. Almeida, "On the interaction between an operating system and web server," Tech. Rep. CS 97-012, Computer Science Department, Boston University, July 1997.

[24] E. P. Markatos, "Main memory caching of web documents," in *Fifth International WWW Conference*, May 1996.

[25] J. C. Hu, S. Mungee, and D. C. Schmidt, "Techniques for developing and measuring high performance web servers over high speed networks," in *Proceedings of the IEEE Infocom'98*, Apr. 1998.

For More Information

An extended version of this paper is available as a Tech. Rep. via *http://www.ele.uri.edu/Research/hpcl/network/apache.html*.