Measuring the Behavior of a World-Wide Web Server

Jussara M. Almeida
(jussara@dcc.ufmg.br)

Virgílio Almeida[†]
(virgilio@bu.edu)

David J. Yates[‡]
(djy@bu.edu)

Depto. de Ciência da Computação,
Universidade Federal de Minas Gerais,
Belo Horizonte,
Minas Gerais 31270-010 – Brazil

Computer Science
Department,
Boston University,
Boston, MA 02215 – USA

Abstract

Server performance has become a crucial issue for improving the overall performance of the World-Wide Web. This paper describes Webmonitor, a tool for evaluating and understanding server performance, and presents new results for a realistic workload.

Webmonitor measures activity and resource consumption, both within the kernel and in HTTP processes running in user space. Webmonitor is implemented using an efficient combination of sampling and event-driven techniques that exhibit low overhead. Our initial implementation is for the Apache World-Wide Web server running on the Linux operating system. We demonstrate the utility of Webmonitor by measuring and understanding the performance of a Pentium-based PC acting as a dedicated WWW server. Our workload uses a file size distribution with a heavy tail. This captures the fact that Web servers must concurrently handle some requests for large audio and video files, and a large number of requests for small documents, containing text or images.

Our results show that in a Web server saturated by client requests, up to 90% of the time spent handling HTTP requests is spent in the kernel. Furthermore, keeping connections open, as required by TCP, causes a factor of 2 increase in the elapsed time required to service an HTTP request. Data gathered from Webmonitor provide insight into the causes of this performance penalty. These results emphasize the important role of operating system and network protocol implementation in determining Web server performance.

1 INTRODUCTION

The quality of networked services like the World-Wide Web (WWW) depends on many factors, including performance, reliability, and security. The overall performance of the Web depends on the performance of its main components; namely clients, the network, and servers. The explosive growth of the Web is placing a heavy demand on servers [Birman *et al*, 1996]. As a result, users see slow response times on the most popular sites, which are overrun by millions of requests per day. Thus, server performance has become a critical issue for improving the quality of service on the World-Wide Web. In order to improve Web server performance, we need to understand how server behavior differs in response to different types of requests, such as requests for small HTML documents, or for large audio and video files. We need to gain insight into server behavior under heavy load in the presence of such heterogeneous requests. In particular, we need to assess the

[†]On sabbatical at Boston University from Universidade Federal de Minas Gerais. Partially supported by CNPq-Brazil.

[‡]This work was supported in part by NSF grants CDA-9529403 and CDA-9623865.

impact of operating system and network protocol implementation on server performance. This suggests the need for quantitative measurements that show how system resources are being utilized when servicing HTTP requests.

Despite the importance of measuring and understanding the behavior of Web servers, there are no freely available performance tools that give detailed information about server behavior. In this paper, we describe and present results from a prototype tool (called *Webmonitor*) that does just this. For an HTTP workload, Webmonitor measures activity and resource consumption, both within the kernel and in HTTP processes running in user space. Webmonitor is implemented using an efficient combination of sampling and event-driven techniques that have low overhead (less than 4%), and therefore does not significantly perturb server behavior. Our initial implementation is for the Apache WWW server running on the Linux operating system.

We demonstrate the utility of Webmonitor by measuring and understanding the performance of a Pentium-based PC acting as a dedicated WWW server. We present results for a workload generated by WebStone [Trent *et al.*, 1995], which is a configurable tool for benchmarking Web server performance, available from Silicon Graphics. We parameterized the server workload generated by WebStone to capture the heterogeneous nature of HTTP requests, using values from [Arlitt *et al.*, 1996]. Specifically, we used a file size distribution with a heavy tail to capture the fact that Web servers must concurrently handle some requests for huge multimedia files and a large number of requests for small HTML and image documents. Such distributions occur in the size of files available at servers, and in files requested by clients [Arlitt *et al.*, 1996, Crovella *et al.*, 1996]. This heterogeneity in workload stresses the limits of the underlying operating system much further than traditional applications [McGrath *et al*, 1995]. One other important characteristic of our workload (and experiments) is that we do not reuse TCP connections for multiple HTTP requests, as described in [Mogul, 1995a] and the Apache documentation [Robinson *et al.*, 1995]. Thus, we open a new TCP connection for every request. We therefore capture the costs of servicing our workload under the "worst case" assumption of being unable to use persistent connections.

We present two new results from data collected using Webmonitor. First, in a Web server saturated by client requests, we find that up to 90% of the time spent handling HTTP requests is spent in the kernel. Second, that keeping TCP connections open causes a factor of 2 increase in the elapsed time required to service an HTTP request. It is necessary to keep TCP connections open (in the TIME_WAIT state) at the server to guard against old data being received by a new connection. Although such problems with the way TCP interacts with HTTP have been pointed out by others [Mogul, 1995c, Mogul, 1995a, Padmanabhan, 1994], we isolate and quantify their impact. Specifically, we show that these lingering TCP connections cause a 33% performance penalty in terms of throughput.

The rest of the paper is structured as follows. Section 2 outlines specific characteristics of the Web that influenced the approach we adopted to measure server behavior. In section 3 we describe the experimental environment that was instrumented and measured, and the workload used to drive our experiments. Section 4 presents an overview of the Webmonitor architecture and important aspects of its implementation. Next, we present and analyze measurements collected by Webmonitor. We then use the tool to measure the behavior of a busy Web server, and discuss the impact of the Web server implementation on performance. Finally, section 5 summarizes the paper.

2 MEASURING A WEB SERVER

The standard performance tools provided by Unix operating systems include *ps*, *vmstat* and *netstat*. In Linux, all of them collect information from /proc filesystem [Welsh, 1994]. Although these tools can provide insight into server behavior, they reflect the performance only from a system-wide standpoint. Furthermore, those standard tools may introduce unbearable overhead during the monitoring of a busy Web server [Cockcroft, 1996].

In order to obtain in-depth information about the server behavior, we also need to collect data at the HTTP server level. HTTP servers usually log per-request information in log files, but that is not enough to gain insight in the way system resources are used to service an HTTP workload. Thus, we decided to build a specific tool to monitor the behavior of Web servers and to measure resource usage. In this section, we describe the guidelines and principles we followed to design a Web server performance monitor.

2.1 Characteristics of Web Servers

As pointed out in [Arlitt *et al.*, 1996, Crovella *et. al.*, 1996, Mogul, 1995a, Mogul, 1995b, Almeida Bestavros *et al.*, 1996], there are several characteristics that distinguish Web servers from traditional distributed systems. The following two characteristics have a profound impact on the behavior of Web servers.

Heavy Tailed Distributions

Recent studies [Arlitt *et al.*, 1996, Crovella *et. al.*, 1996] have shown that file sizes in the World-Wide Web exhibit heavy tails, including files stored on servers, files requested by clients and transmitted over the network. A heavy-tailed distribution (e.g., Pareto) is given by $P[X > x] \sim x^{-\alpha}$, as $x \to \infty$ and $0 < \alpha < 2$. Theoretical heavy-tailed distributions have infinite variance, which, in practical terms, means that very large observations are possible with non-negligible probability. In [Crovella *et al.*, 1996], the authors surveyed a number of WWW servers in the Internet and found evidence of heavy-tailed distributions of sizes of files on the servers. One possible explanation is the presence of large multimedia files that contribute to increase the tail of file size distribution.

Short-lived Processes

Most HTTP server implementations use a new TCP connection for almost every request. Several references [Arlitt *et al.*, 1996, Crovella *et al.*, 1996] report that over 90% of client requests are for small HTML or image files. The combination of these facts explains a common phenomenon that has been observed during the operation of busy Web servers: the creation of a large number of short-lived processes [Mogul, 1995a, Mogul, 1995b]. This brings new challenges to some operating systems that are not tuned for handling a large number of short-lived processes. Short-lived processes also represent new problems for performance monitoring. Although UNIX provides accurate measurements for processor usage by processes of moderately long duration, the authors in [Somin *et al.*, 1996] point out the problems in trying to measure CPU time used by short-lived individual processes.

2.2 Measurement Approach

The fundamental characteristics of a good measurement tool are low overhead, low interference in the system being measured, and high accuracy. We address these characteristics in the design and implementation of Webmonitor.

Although monitors can provide a great deal of useful data, there are problems with the use of their data for performance modeling. Thus, Webmonitor was designed to provide data for analytical models also. The basic input data required by queueing network models are service demands of a request at a server [Menasce *et al.*, 1994]. Those demands specify the total amount of service time required by a request during its execution at each major component of the server. It is worth mentioning that service demand refers only to the time a request spends actually receiving service. It does not include waiting times. Webmonitor was designed to provide this information, which can then be used to derive the basic data required by analytical and simulation queueing models.

In this section we show the features of Webmonitor that take advantage of World-Wide Web workload characteristics to achieve low overhead and high accuracy.

Monitoring Techniques

Webmonitor uses a combination of sampling and event-driven techniques to collect different levels of information about the operation of a Web server. Sampling-based measurement is used to read counters that are maintained by the kernel. Those counters provide system-level information (e.g., resource utilization, interrupt rates, etc.) as well as network statistics. Because events occur within different modules in a Web server, our monitor supports the concept of different sampling intervals, that are adjusted to the nature of the information being monitored. However, the choice of sampling intervals always represents a trade-off between accuracy and overhead. To do sampling in an efficient way, we made some modifications to the Linux kernel, because of the high volume of requests that would imply high overhead.

Sampling is an inadequate technique to trace the execution of every HTTP request in user space. Thus, for monitoring the execution of every request in the HTTP processes, Webmonitor uses an event-driven technique that required the instrumentation of the server. However, it is important to note that this instrumentation was designed to be lightweight to minimize its effect on server processing.

Classes of Requests

Although it would be desirable to have detailed execution information about each individual request, it is unfeasible, in terms of overhead, to keep track and record this quantity of information. This is especially true for busy Web servers that are overloaded by requests. A possible solution would be to simply accumulate the execution information for all requests and to calculate average values for the measurements. However, as we saw earlier in this paper, requests for documents at Web servers follow heavy tailed probability distributions, that have very large variance. Thus, average results for the whole population of requests would have no statistical meaning.

As a compromise to keep overhead as low as possible without impairing the accuracy and significance of the measurements, we catagorized requests into a small number of classes. A class is defined by a range of file sizes, and these ranges are chosen to reflect a heavy tailed distribution of file sizes on the server. Thus, each class comprises requests that are similar with respect to the size of the files they retrieve. As a result, we group together requests of similar behavior in terms of resource usage, which helps reduce the variance of the collected data.

3 EXPERIMENTAL SETUP

This section explains in detail the WWW server which we used in our experiments. We describe the workload, hardware, and software used to perform the measurements and collect the performance data.

The operating system used is Linux version 2.0.0, which is distributed under the terms of GNU General Public License [Welsh, 1994]. The server software is Apache, version 1.1.1, a public domain HTTP server [Robinson *et al.*, 1995].

Apache was originally based on code and ideas found in NCSA HTTP server [McGrath *et al.*, 1995]. It is "**A PA**t**CH**y server", in the sense that it was based on some existing code and a series of "patch files". Apache can run in two different modes: from the `inetd` system process or, in standalone mode. The main disadvantage of running an HTTP server from `inetd` is that, for each HTTP connection received, a new copy of the server is started from scratch; after the connection is complete, this program exits. Thus, there is a high per-connection overhead. Standalone is therefore the most common mode of operation, since it is far more efficient. The server is started once, and services all subsequent connections.

Another interesting point worth mentioning concerns the management of the HTTP processes. Apache maintains a pool of child server processes to handle incoming requests. On startup, a master server process spawns a pre-defined number of child processes and as the load in the server increases, new processes are spawned and included in this pool. The master process periodically checks the number of idle child processes and dynamically adapts this number to the load it sees. There are pre-defined limits (lower and upper bounds) to the number of idle processes. Besides this, there are also upper bounds for the number of requests each child is allowed to process before it dies and on the total number of child processes running, that is, a limit on the number of clients that can simultaneously connect to the server.

Our Apache server was configured to run in standalone mode. The number of KeepAlive requests per connection [Robinson *et al.*, 1995] was set to 0 (only one HTTP request was serviced per connection). The lower and upper bounds in the number of idle processes were set to 5 and 10, respectively; and the number of requests a child process serves before dying was set to 30. Our hardware platform was an Intel Pentium 75MHz system, with 16 Megabytes of main memory and a 0.5 Gigabyte disk. It has a standard 10 Megabit/second Ethernet card. Linux was installed on the disk on a partition of 416 Megabytes, and a partition of 36 Megabytes was allocated for swap space.

To generate a representative WWW workload, we used WebStone [Trent *et al.*, 1995] (version 2.0), which is an industry-standard benchmark for generating HTTP requests. WebStone is a configurable client-server benchmark for HTTP servers, 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.

WebStone is a distributed, multi-process benchmark, where a master process spawns, local or remotely, a pre-defined number of client processes. Each client process generates requests to the server and collects the performance statistics. After all clients finish running, the master process collects the client's statistics and calculates the overall server performance during the execution of the workload. In our experiments, the client processes were spread over three machines: two SparcStation 20 (128 and 256 megabytes of main memory and operating systems SunOS 4.1.4 and 5.5) and one SparcStation Ultra (128 megabytes of main memory and SunOS 5.5) In order to generate load for a WWW server, client processes successively request files from the server, as fast as the server can

Table 1 Characteristics of an HTTP Workload

		File size (KBytes)		Access probability	
Item	**Number of files**	**Total**	**Average**	**Total**	**Average**
HTML	24	180	7.5	0.192	0.008
Images	29	385	13.28	0.754	0.026
Sound	20	3580	179	0.05	0.0025
Video	4	9216	2304	0.004	0.001

answer the requests. A new request is sent out to the server right after a client receives the answer from a previous request. The main performance measures collected by WebStone are latency and throughput. The former represents the response time to complete a request, viewed from the client side. Throughput is measured in connections per second and also in bytes transferred per second.

The WebStone workload is defined by the number of client processes and by the configuration file that specifies the number of files, their size and access probabilities. Table 1 gives baseline information for the HTTP workload used in our experiments. The parameters that define the workload are representative of the kinds of workload typically found in busy WWW servers [Arlitt *et al.*, 1996]. It is worth noting that the set of files in this workload consumes 82% of physical memory. Furthermore, once the kernel and HTTP processes are also present in memory, we observe significant disk activity in our experiments.

4 ARCHITECTURE OF THE MONITOR

Figure 1 depicts an overview of Webmonitor. The monitor can be seen as a combination of two main components that operate at different levels of the system and collect performance data using different techniques. This division in based on the interaction between the monitor and system, the technique of instrumentation used and the nature of the data collected. The Kernel Module runs independently of the Web server and collects information about the operating system as a whole. The code of the Server Module is actually linked with the server code, and therefore runs as part of the server. It collects information about server performance during the handling of HTTP requests.

4.1 The Kernel Module (KM)

The Kernel Module (KM) collects resource usage data, not only from a system-wide standpoint but also from the Web server viewpoint. The information collected is: processor utilization, disk activity, paging activity, and interrupt rates. This module also collects information about network activity, which is divided into two groups. The first one refers to statistics on communication activities through the Ethernet interface, such as the number of packets transmitted or received, number of errors that occurred during transmission or reception. The second group provides information about the number and state of TCP connections to the HTTP port in the server. The TCP state information is useful for understanding the "lifetime" of connections in the server.

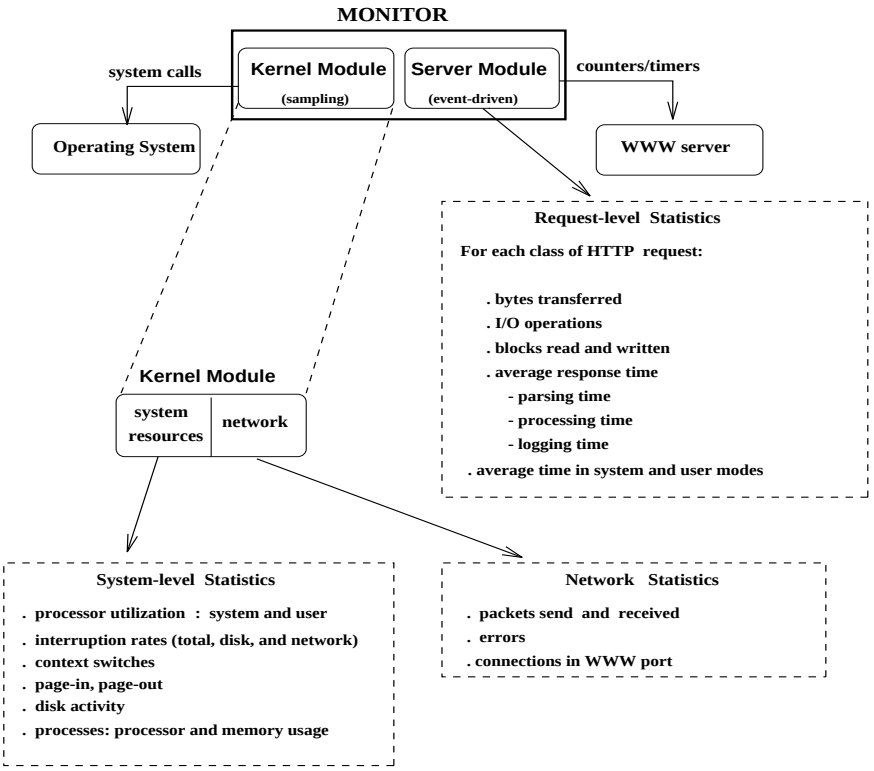

Figure 1 Overview of the Webmonitor

In addition to the three types of system-wide information activity described above, KM also obtains information about certain processes. The information is basically CPU and memory utilization. It also collects the total number of copies of each monitored program (**started processes**) and the number of copies waiting for run time (**running processes**). In our experiments, we chose to monitor the HTTP processes and the kernel processes responsible for swapping and buffer cache management. However, since our results show that the vast majority of system resources are consumed by the HTTP processes, we only present results for these processes.

Usually the Linux kernel keeps performance data internally. They can be read by user programs through the /proc filesystem [Welsh 1994]. This is a "virtual file system", in the sense that its contents are not located on disk but in memory. A read of any file below /proc causes data in the kernel to be copied to memory in user space. This information is actually copied as a sequence of ASCII characters. Thus, to find specific data, it is necessary to parse a string for a specific keyword and then read one or more numeric values.

There is one important disadvantage to using /proc to gather kernel activity information. If one needs to gather information scattered throughout several kernel data structures, one must perform multiple reads (each of which is a system call), or read very large blocks of data out of the kernel. Both of these alternatives are very expensive. This overhead of reading /proc, to get specific but scattered information, is the main reason we decided to implement the KM using four new system calls.

The information gathered by the KM is collected through four system calls that summarize and return specific information about kernel activity in a single buffer. The KM system calls are as follows:

- **my_get_kstats**: returns information about processor utilization, disk activity, paging activity and interrupt rates.
- **my_get_procstats**: returns cpu and memory utilizations for each process with a given command name.
- **my_get_netstats**: returns the number of packets transmitted and received and the number of errors occurred in the network interface.
- **my_get_connstats**: returns the number of connections in each TCP state connected to a given TCP port.

The KM runs as a group of two to four processes, periodically collecting information through the system calls described above. The number of samples, the TCP port to be monitored, the number of different programs to be monitored, the name of them and the number of KM processes spawned are parameters specified in a configuration file [Almeida Almeida *et al.*, 1996].

4.2 The Server Module (SM)

The Server Module (SM) is responsible for collecting information about server performance during the handling of HTTP requests. It is implemented as a library of routines compiled and linked with the server code. Calls to specific routines were inserted at appropriate points in the server code. Instead of being based on sampling, like the KM described in the last section, the SM collects informating based on a trace of events that occur during the handling of a single request. The data collected are: bytes transmitted, connections established, read and write operations, and number of blocks read and written during the handling of the request. Another important piece of information is the processing time at the server to handle a request. The time measured by the SM begins with the establishment of a connection and ends when the server (i.e., HTTP process) is ready to handle the next request. It is broken into three components, which are measured in processor time and in elapsed time. *Parsing time* is the interval that begins just after the establishment of the connection and ends when the header of the request has been parsed and is ready to be processed. *Processing time* is the time spent actually processing the request. It does not include the server logging time. It accounts for the time spent reading the URL (Uniform Resource Locator) and the time needed to move the file from memory or disk to the network. *Logging time* is the time spent performing standard HTTP logging. After logging, a server process is ready to handle a new request.

Unfortunately, the Linux timing routines are not accurate enough to account for the three components of the execution time of a short request. The timing resolution is on the order of 10 milliseconds [Welsh, 1994]. In order to measure parsing, processing, and logging times with greater accuracy, we implemented a "stopwatch" scheme using the `gettimeofday` routine, that returns the elapsed seconds and microseconds since a predefined date. This resolution is because `gettimeofday` reads the time directly from the hardware timer. In order to be timed using a stopwatch, a process must call a system routine to include itself in a *CPU Monitored Processes Table*, located in kernel memory. This routine returns the entry allocated in the table for that process. There are also system calls to *Start* and *Stop* the time accounting. To discount the time that the CPU was used by processes other than the one being monitored, an entry of the *CPU Monitored Processes Table* also contains the time between *Start* and *Stop* spent servicing other processes [Almeida Almeida *et al.*, 1996].

A similar scheme was implemented in order to collect per-process disk activity information. It creates a *Disk Monitored Processes Table*, where appropriate information is kept.

To be monitored, a process must allocate an entry in this table through a system call. Every time a disk request from a process being monitored is served, the number of read or write operations and the blocks transfered are registered in its entry in this table.

Each server process collects the statistics described above for the requests that it services. In addition, the SM incorporates the concept of *request classes*. Each request is categorized into one of several predefined classes depending on the size of the file requested. The classes are defined in a configuration file specifying the maximum file size for each class. The statistics collected by a server process are separated by class. Thus, while handling a request, a server updates the counters associated with the class of the request being serviced. In this manner, the SM generates cumulative information for each class and each HTTP server process. To keep overhead low, this information is written to disk by the server processes after 10 requests have been served. After data collection is complete, these cumulative values can be processed to generate other statistics such as averages, variances, etc.

4.3 Monitor Overhead

One of the main concerns in the design of the WWW server monitor was to keep overhead as low as possible. The response time (in seconds) and throughput (conn/s and Mbits/s) measured by WebStone for the server with the monitor (KM and SM modules) were 1.74, 17.35 and 3.91, respectively. Without the monitor, WebStone measured 1.69, 17.96 and 4.02, for the same workload. Thus, the overhead introduced by the monitor is less than 4% for all three measures. We also compared the cost of using our system calls against the cost of obtaining the same information through the /proc filesystem. Compared to Webmonitor system calls, collecting the same data via /proc is between 5 and 200 times more expensive [Almeida Almeida *et al.*, 1996].

5 RESULTS

Recall that one of the main design goals of our WWW server performance monitor is to understand how time is spent servicing HTTP requests, and how different components of the server software are utilized. The KM addresses this goal by measuring the CPU user and system time, and the rate at which different kernel services are invoked (e.g., read calls per second). The SM addresses this goal by measuring the CPU utilization and latency of servicing requests, as well as tracking per-connection use of some kernel services (e.g., read calls per connection).

We demonstrate the utility of our WWW server performance monitor at the most interesting operating point of the server – when it has just become saturated. To determine the saturation point, we ran experiments varying the number of WebStone clients that communicate with the server. Our results, average values from 3 experiments, are for 30 clients, which cause both the CPU and memory of the server to be utilized at levels greater than 90%. We discuss these results for the server module first, then describe results from the kernel module, and then tie them together. Finally, we present results for experiments where we change the Linux TCP implementation to *not* keep connections open at the server. Comparing these results with our original results shows the effect that keeping TCP connections open has on server performance.

5.1 Server Module Results

Table 2 shows server module (SM) measurements for the three different classes of requests. Recall that these request classes correspond to different file sizes that span a heavy-tailed distribution. Furthermore, each class is representative of an object "class," as in Table 1. Class 1 requests (for HTML and image documents) are for small files; they have a mean size of 12.1 KB and make up the vast majority of the requests (i.e., 94.6%). Class 2 requests (for audio files) are moderate in size and amount to 5% of requests. Class 3 requests (for video clips) are large (2.3 MB on average) and make up only 0.4% of the workload. The

Table 2 Server Module Results for 30 Clients.

	Class 1	**Class 2**	**Class 3**
conn/s	16.40	0.88	0.07
Mbits/s	1.55	1.17	1.19
reads/conn	0.03	2.67	34.67
CPU parsetime(ms)/conn	4.78	4.81	3.81
CPU processingtime(ms)/conn	18.75	150.56	2231.35
CPU logtime(ms)/conn	5.28	6.99	9.10
elapsed parsetime(ms)/conn	23.45	21.80	6.20
elapsed processingtime(ms)/conn	155.81	3789.95	60578.90
elapsed logtime(ms)/conn	774.31	940.35	827.42

most interesting result in Table 2 lies in the last six rows, which show the processor time and the elapsed time of the three different phases of execution of an HTTP request. These rows show that in most cases the majority of the time spent servicing an HTTP request is spent moving the requested URL from the filesystem to the network (i.e., processing the parsed request). This is true of CPU time for all three request classes in our workload. Furthermore, the elapsed processing time also dominates the elapsed parse and logging times for moderate and large (Class 2 and 3) requests. The CPU time and elapsed time for processing requests increases by three orders of magnitude as the mean file size for the three classes does also. The other measurements shown in Table 2 which show the same increase are the read calls per connection. This suggests that disk activity explains the increase in elapsed time for processing large requests, as one would expect. One other interesting result in Table 2 is the distribution of network bandwidth among the three request classes. Note that even though the connections per second rate decreases with class number (and requested file size), the bandwidth that each class consumes on the network is about the same (i.e., between 1 and 1.6 Mbps). This is due to the heavy-tailed nature of the file size distribution.

The results from Table 2 suggest that most of the CPU time consumed by the HTTP processes is spent in the kernel. In other words, the task of moving the requested URL from the filesystem to the network is the most expensive part of handling a request. Since both the filesystem and the networking code are in the kernel, one would expect time spent in the kernel to be greater than time in user space. We tested this hypothesis by instrumenting the HTTP processes to call `getrusage` after every 10 requests, and report

Table 3 Kernel Module Results for **my_get_kstats** and 30 clients

cpu_user(%)	9.00	**pageins/s**	137.42
cpu_sys(%)	90.65	**pageout/s**	7.94
cpu_idle(%)	0.35	**interrupt/s**	1011.22
reads/s	5.17	**net_interrupt/s**	620.43
writes/s	4.90	**disk_interrupt/s**	289.90
context switch/s	51.66		

Table 4 Kernel Module Results for **my_get_procstats** and 30 clients

cpu(%)	93.75	**started processes**	27.55
mem(%)	102.61	**running processses**	25.01

the user and system time per connection, for the duration of the experiment. These results show that our HTTP processes consume an average of 50 msec of CPU time in the kernel per connection, compared with only 5.2 msec in user space. We'll see later that the kernel module results also demonstrate this high (i.e., 10:1) ratio of system CPU time to user CPU time, for the WWW server as a whole.

5.2 Kernel Module Results

Table 3 shows kernel module (KM) measurements for the workload described above. Recall that KM measures only kernel-level statistics such as overall CPU user and system time, and the rate at which different services are invoked. It therefore does not separate its measurements according to request class. The most interesting result in Table 3 is that the ratio of system time to user time is high, and is approximately the same as for the HTTP processes monitored by the SM, i.e., 10:1. Within the time spent in the kernel, it is also important to note the relative frequency of certain kernel operations. For example, there are over 100 page-in's, network interrupts, and disk interrupts per second. There are several read calls performed per second. However, there are also a significant number of corresponding write operations per second. These are presumably due to paging activity and logging of HTTP requests.

We have seen a correspondence between SM and KM statistics looking at Table 2 and then Table 3. We also wanted to show a correspondence in the reverse direction. Table 4 shows aggregate process statistics for the HTTP processes, measured in the kernel. Note that the CPU is over 90% utilized, and that memory utilized by the HTTP processes alone is over 100% (which indicates paging activity) . This explains why the CPU user and system times for the HTTP processes (measured by SM) and for the system as a whole (measured by KM) agree. The number of running processes suggests that Apache's process management requires 25 processes to service 30 concurrent connections, given our workload. Finally, the measurements obtained by KM concerning network statistics reported no errors in network interface during the experiments, which is consistent with WebStone results that also reported no errors on the client side.

Table 5 KM Results for Workload with Smaller Files

cpu_user(%)	9.87
cpu_sys(%)	87.07
cpu_idle(%)	3.06
reads/s	0.79

Table 6 SM Results for Workload with Smaller Files

user time(ms)/conn	23.97
system time(ms)/conn	2.83
(aggregate) conn/s	25.74
(aggregate) Mbits/s	3.45

We also wanted to show that our main conclusions still held when the set of files being requested at the server was sufficiently small to reduce disk activity. A new workload was obtained by dividing the file sizes presented in Table 1 by a factor of 4, but keeping the same number of clients. The results for this workload are shown in Tables 5 and 6. In Table 6, the SM request class sizes are also scaled down by a factor of 4. These results show that, despite reduced disk activity, the ratio of system time to user time remains very high. Both KM and SM results show that over 89% of the CPU time is spent in the kernel. It is also interesting to note the higher connection rate, due to a much faster handling of each request.

The validation of the results collected by KM and SM was done through comparison with similar measurements obtained through the /proc filesystem and WebStone, respectively. The differences between them are less than 1% [Almeida Almeida *et al.*, 1996].

5.3 Effect of Keeping TCP Connections Open

We wanted to use Webmonitor to measure the effect of keeping TCP connections open on our Web server. Recall that this is a requirement of TCP, to guard against old data being received by a new connection. To isolate this effect, we reproduced the experiments described above, but changed Linux's TCP implementation to close connections without spending any time in the TIME_WAIT state. Although such a TCP implementation is not "legal," this modification allowed us to show the effect of keeping connections open on server behavior. In a legal implementation, the TIME_WAIT state is entered to catch and discard packets from a closed connection, that were retransmitted by a client. The usual holding time in this state is 60 seconds, after which the connection is closed (put in the TCP_CLOSE state). It has been observed by others [Mogul, 1995a, Mogul, 1995c, Padmanabhan *et al.*, 1994] that the holding time in the TIME_WAIT state is a possible performance problem for WWW servers, however, we are the first to quantify this and give some insight into possible causes.

Table 7 gives the average number of connections seen in different TCP states. Although TCP actually has 11 states, the number of connections in the other 8 states was zero or negligible. The most interesting number in Table 7 is the large number (over 900) connections in the TIME_WAIT state, when its holding time is 60 seconds. These results are consistent with those in [Mogul, 1995a, Mogul, 1995c]. Fortunately, because of the large number of TCP connections that may be open at the same time, Linux uses a hashed lookup table with a single entry cache to store its connection desriptors. Such an implementation guards against connection descriptor lookup times that increase linearly with the number of open connections (a common mistake in older TCP implementations). It is also interesting to note that more time is spent in the closed state (TCP_CLOSE), than in

the state where the connections are actually performing useful work (the ESTABLISHED state).

Table 7 Number of Connections in TCP States (KM)

TCP State	TIME_WAIT = 0	TIME_WAIT = 60 sec
ESTABLISHED	29.14	29.74
TIME_WAIT	0	941.84
CLOSE	64.45	35.03

Table 8 Server Module Results for TIME_WAIT of 60 with 30 Clients

TIME_WAIT = 60 sec	Class 1	Class 2	Class 3
conn/s	16.40	0.88	0.07
Mbits/s	1.55	1.17	1.19
reads/conn	0.03	2.67	34.67
CPU parsetime(ms)/conn	4.78	4.81	3.81
CPU processingtime(ms)/conn	18.75	150.56	2231.35
CPU logtime(ms)/conn	5.28	6.99	9.10
elapsed parsetime(ms)/conn	23.45	21.80	6.20
elapsed processingtime(ms)/conn	155.81	3789.95	60578.90
elapsed logtime(ms)/conn	774.31	940.35	827.42

Table 9 Server Module Results for TIME_WAIT of 0 with 30 Clients

TIME_WAIT = 0	Class 1	Class 2	Class 3
conn/s	24.64	1.25	0.10
Mbit/s	2.33	1.66	1.87
reads/conn	0.02	2.24	33.91
CPU parsetime(ms)/conn	2.41	2.34	2.41
CPU processingtime(ms)/conn	15.84	101.52	1373.71
CPU logtime(ms)/conn	6.09	7.07	7.05
elapsed parsetime(ms)/conn	8.62	6.26	5.83
elapsed processingtime(ms)/conn	77.89	2519.07	37417.10
elapsed logtime(ms)/conn	559.37	559.42	530.52

To understand the impact this large number of TIME_WAIT connections has on server

performance, we first looked at results from the SM. Tables 8 and 9 show SM results for 30 clients using a TIME_WAIT time of 60 seconds and 0, respectively. The results for latency and throughput (conn/s and Mbit/s) show a dramatic difference in performance. Having a TIME_WAIT time of 60 seconds makes all the work that a server performs (i.e., parsing, processing and logging an HTTP request) take longer. This is true in terms of both CPU time and elapsed time. For example, the CPU time to process a large (i.e., Class 3) request is two times greater for a TIME_WAIT time of 60 seconds than for a TIME_WAIT time of 0.

As a consequence of the increase in latency when the TIME_WAIT time is set to 60, throughput decreases significantly. The performance penalty for the server can be seen by a 33% higher number of connections serviced (and Mbits sent) per second, when the TIME_WAIT time is 0. It is also interesting to examine the resources consumed at the server. Our KM results for both configurations (not presented here) show that the consumption of all resources is roughly the same. Both CPU and memory utilizations are over 90%, indicating that the server is saturated in both experiments. Furthermore, roughly the same number of HTTP processes are used to handle the higher request rate when no time is spent in the TIME_WAIT state.

These results clearly show that the impact of the TIME_WAIT holding time is twofold. First, although roughly the same number of HTTP processes are active at the same time, these processes are able the handle a lower number of requests. Second, that these processes consume more CPU time to serve each request. This is, of course, only part of the answer to the larger question of whether memory, I/O, or the CPU is the bottleneck for WWW servers.

6 CONCLUSION

Server performance has become a crucial issue for improving the overall performance of the World-Wide Web. This paper describes Webmonitor, a tool for evaluating and understanding server performance, and presents new results for a realistic workload. These results emphasize the important role of operating system and network protocol implementation in determining Web server performance.

Webmonitor measures activity and resource consumption, both within the kernel and in HTTP processes running in user space. Webmonitor is implemented using an efficient combination of sampling and event-driven techniques that exhibit low overhead (less than 4%). We demonstrate the utility of Webmonitor by measuring and understanding the performance of a Pentium-based PC acting as a dedicated WWW server. Our workload, generated by WebStone, uses a file size distribution with a heavy tail. This captures the fact that Web servers must concurrently handle some requests for huge files and a large number of requests for small files.

Our results show that in a Web server saturated by client requests, up to 90% of the time spent handling HTTP requests is spent in the kernel. Furthermore, keeping connections open, as required by TCP, causes a factor of 2 increase in the elapsed time required to service an HTTP request. This increase in latency (caused by a high number of connections in the TIME_WAIT state) is accompanied by a 33% reduction in server throughput.

Although this paper provides an important understanding of World-Wide Web server behavior under heavy load, the picture is far from complete. There is still the question of whether memory, I/O, or the CPU is the bottleneck for Web servers. The answer to this question will probably depend on the nature of the workload, however, there will continue to be a demand for server architectures that perform well for heterogeneous workloads.

This suggests the need for new operating system and network protocol implementations that are designed to perform well when running on Web servers.

ACKNOWLEDGEMENTS

We would like to extend special thanks to Pei Cao and Gideon Glass at the University of Wisconsin for insightful discussions about this work. We would also like to thank Jeff Mogul at DEC WRL, Erich Nahum at IBM Watson, Wagner Meira at the University of Rochester, and the anonymous reviewers for improving the content and presentation of this paper.

REFERENCES

Almeida, J. M., Almeida, V. and Yates, D. (1996), Measuring the behavior of a World-Wide Web Server, Technical Report 96-025, Boston University.

Almeida, V., Bestavros, A., Crovella, M. and Oliveira A. (1996), Characterizing reference locality in the WWW, *Proceedings of IEEE-ACM PDIS'96.*

Arlitt, M. and Williamson, C. (1996), Web server workload characterization, *Proc. of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

Birman, K. and Renesse, R. (1996), Software for reliable networks, *Scientific American.*

Cockcroft, A. (1996), Watching your web server, *SunWorld Online*, URL: http://www.sun.cm/sunworldonline/swol-03-1996/.

Crovella, M. and Bestavros, A. (1996), Self-similarity in world wide web traffic: Evidence and possible causes, *Proc. of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

Lai, K. and Baker, M. (1996), A performance comparison of UNIX operating systems on the Pentium. *Proceedings of the 1996 USENIX Conference*, San Diego, CA. USENIX.

McGrath, R., Kwan, T. and Reed, D. (1995), NCSA's world wide web server: Design and performance, *IEEE Computer.*

Menasce, D., Almeida, V., Dowdy, L. (1994), *Capacity Planning and Performance Modeling*, Prentice Hall, Englewood Cliffs.

Mogul, J. C. (1995), Network behavior of a busy Web server and its clients, Research Report 95/5, DEC Western Research Laboratory.

Mogul, J. C. (1995), Operating system support for busy Internet servers, *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems.*

Mogul, J. C. (1995), The case for persistent-connection HTTP, In *SIGCOMM Symposium on Communications Architectures and Protocols*, **299-313**, Cambridge, MA. ACM

Padmanabhan, V. N. and Mogul, J. C. (1994), Improving HTTP latency, In *Proceedings of Second WWW Conference '94: Mosaic and the Web*, **995-1005**, Chicago, IL.

Robinson, D. and the Apache Group (1995), *APACHE – An HTTP Server, Reference Manual,*, URL: http://www.apache.org.

Somin, Y., Agrawal, S. and Forsyth, M. (1996), Measurement and analysis of process & workload CPU times in UNIX environments, *Proceedings of the CMG'96.*

Trent, G. and Sake, M. (1995), *WebStone: The First Generation in HTTP Server Benchmarking*, URL: http://www.sgi.cm/Products/WebFORCE/WebStone/paper.html

Welsh, M. (1994), *The Linux Bible*, Yggdrasil Computing Incorporated, $2^{\underline{nd}}$ edition.