

# Measuring the Behavior of a World-Wide Web Server

Jussara Almeida\*

Virgílio Almeida<sup>†</sup>

David J. Yates <sup>‡</sup>

(jussara@dcc.ufmg.br)

(virgilio@bu.edu)

(djay@bu.edu)

Technical Report CS 96-025

October 29, 1996

## 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, over 90% of the time spent handling HTTP requests is spent in the kernel. Furthermore, keeping TCP connections open, as required by TCP, causes a factor of 2-9 increase in the elapsed time required to service an HTTP request. Data gathered from Webmonitor provide insight into the causes of this performance penalty. Specifically, we observe a significant increase in resource consumption along three dimensions: the number of HTTP processes running at the same time, CPU utilization, and memory utilization. These results emphasize the important role of operating system and network protocol implementation in determining Web server performance.*

---

\*Depto. de Ciencia da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, MG 30161, Brazil.

<sup>†</sup>On sabbatical at Boston University from Universidade Federal de Minas Gerais. Partially supported by CNPq-Brazil

<sup>‡</sup>Computer Science Department, Boston University, Boston, MA 02215, USA.

## 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 [5, 10]. 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 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 3%), 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 [21], 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 [4]. 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 requested at servers, and in files requested by clients [3, 8]. This heterogeneity in workload stresses the limits of the underlying operating system much further than traditional applications [20]. One other important characteristic of our workload (and experiments) is that we do not reuse TCP connections for multiple HTTP requests, as described in [16] and the Apache documentation [18]. 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 90% of the time spent handling HTTP requests is spent in the kernel. Second, that keeping TCP connections open causes a factor of 2-9 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 [16, 14, 17], we isolate and quantify their impact. Specifically, we show that these lingering TCP

connections have three noteworthy effects on the server: a factor of 7 increase in the number of HTTP processes running at the same time, a 70% increase in CPU utilization, and a 150% increase in memory utilization.

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 [22]. 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. In [7], the author notes that in a highly loaded Web server (100 http/sec) *netstat -s* took several seconds to run and stalled the server for that time period.

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 for performance analysis. Usually, log files contain only the time the request was received, the file requested, the number of bytes served, and the number of errors occurred during the handling of the requests. That information 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 [3, 8, 14, 15, 1], 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.

#### 2.1.1 Heavy Tailed Distributions

Recent studies [3, 8] 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  $x \rightarrow \infty$  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 [8], 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.

### 2.1.2 Short-lived Processes

Most HTTP server implementations use a new TCP connection for almost every request. Several references [3, 8, 2] report that over 90% of client requests are for small HTML or image files. For instance, reference [2] examines over 2.6 million HTML documents in the Internet and shows that the mean size of the documents is 4.4 KB. 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 [14, 15]. 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 [19] point out the problems in trying to measure CPU time used by short-lived individual processes.

## 2.2 Measurement Principles

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 [13]. 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.

## 2.3 Measurement Approach

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.

### 2.3.1 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. Due to the large number of TCP connections in a busy server, it is desirable to sample the HTTP port quite often. On the other hand, disk activity counters can be sampled less frequently in the same interval. 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.

Sampling is an inadequate 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.

### 2.3.2 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 categorized 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 Environment

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.

### 3.1 The Server System

The operating system used is Linux version 2.0.0, which is distributed under the terms of GNU General Public License [22]. The server software is Apache, version 1.1.1, a public domain HTTP server [18].

Apache was originally based on code and ideas found in NCSA HTTP server [20]. It is “**A PAtCHy** server”, in the sense that it was based on some existing code and a series of “patch files”. It supports the notion of optional modules, that are compiled and linked to the main code. These modules are responsible for implementing features such as cgi scripts and proxy server support, and authentication and access checking. 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. In other words, it tries to maintain enough child processes to handle the current load, plus a few spare servers to handle transient load spikes. There are pre-defined limits (lower and upper bounds) to the number of idle processes. New processes are spawned if the lower bound is reached, and, in case of a very high number of idle processes, some of them die off. 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. Apache incorporates some features of HTTP 1.1 since it can accept more than one HTTP request per connection [16].

Our Apache server was configured to run in standalone mode. The number of KeepAlive requests per connection [18] 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. On startup, we spawned 5 child processes. 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.

### 3.2 Workload

To generate a representative WWW workload, we used WebStone [21] (version 2.0.0), 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. Client processes and server run on different machines. In our experiments, the client processes run on a SparcStation 20 with 256 megabytes of main memory and operating system SunOS 5.4. The number of client processes is limited only by the available memory in the client machines. In order to generate load for a WWW server, client processes successively request pages and files from the server, as fast as the server can answer the requests. 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 pages, their size and access probabilities. A request for a page means a request for each file that makes up the page. 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 [4].

Item	Number of files	File size (KBytes)		Access probability	
		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

Table 1: Characteristics of an HTTP Workload

## 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 is based on the interaction between the monitor and system, the technique of instrumentation used and the nature of the data collected [9]. 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.

In addition to the three types of system-wide information activity described above, KM also obtains information about certain processes. The information is CPU and memory utilization, and the number of major page faults. 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 [22]. 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

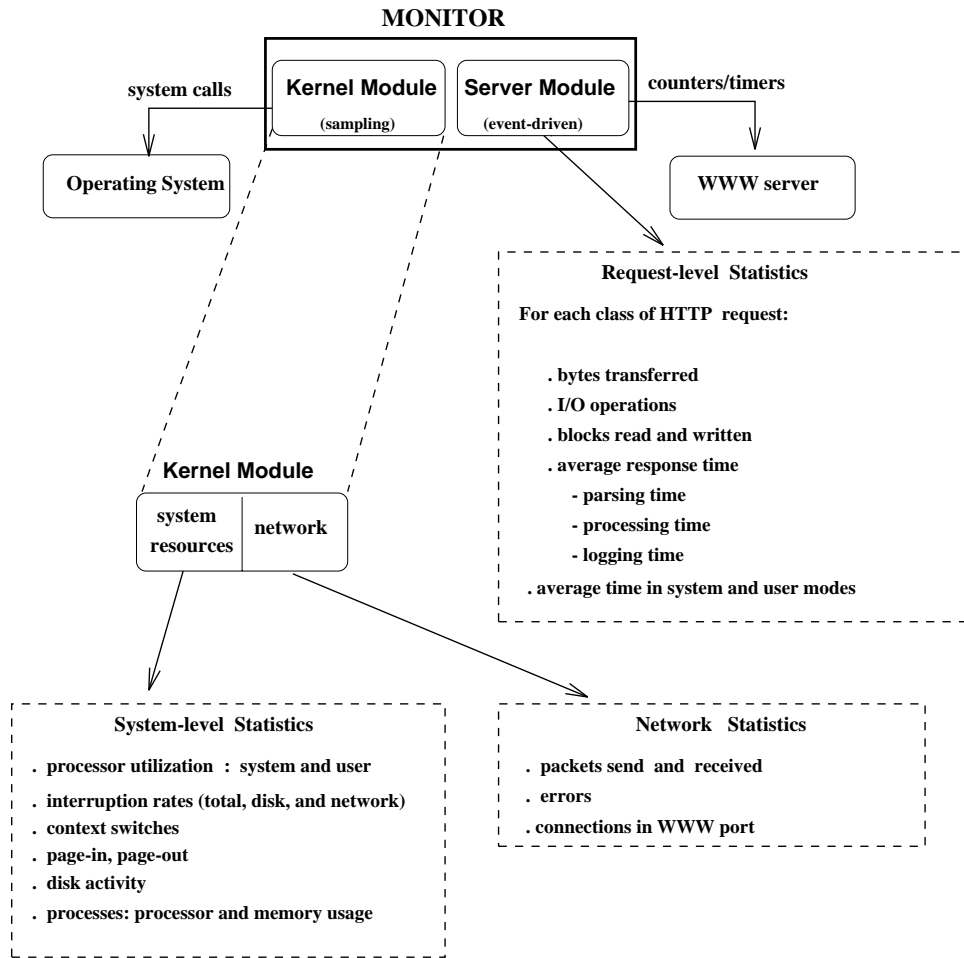


Figure 1: Overview of the Webmonitor

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. All information is returned as a cumulative value since the last system boot, however, each buffer contains a field called *uptime* that records the time since the last boot. Therefore, it is possible to compute rates and percentages from the data returned by these system calls. This processing is done after data collection in our experiments in order to minimize overhead during data collection. 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 and the name of them are parameters specified in a configuration file. The performance data are divided into four categories depending on the system call: `KERNEL_STATS`, `PROC_STATS`, `NET_STATS` and `CONN_STATS`. For each group, it is possible to specify if it's enabled (i.e., the information is collected) and the interval of sampling. If both `KERNEL_STATS` and `PROC_STATS` are enabled and their intervals of sampling are equal, a unique process is spawned to collect both group of information. If not, one process is spawned for each group. The same is true for the other two groups of information. One log file is created for each group of information.

## 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 information 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 of 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 [22]. 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 watch. The former starts the time accounting and the latter returns the time the process spent using the CPU since the last time *Start* was called. 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* has three other fields, besides the starting and stopping times:

- *Begin*: The last time this process left the CPU in a context switch.
- *End*: The last time this process got the control of the CPU.
- *Time*: Total time between *Start* and *Stop* spent servicing other processes.

The time that a process leaves the CPU is written into *Begin*. The next time it runs is written into *End*, and the difference between these two values is accumulated in *Time*. Thus, the *Stop* call computes the difference between the stop and start times, subtracts from it the accumulated value in *Time* and returns the result.

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 transferred 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 *resource 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. Table 2 summarizes the main measures obtained by Webmonitor.

### 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. Table 3 displays response time and throughput (conn/s and Mbits/s) measured by WebStone for the server with and without the monitor. These results are the average of three experiments. Note that the overhead introduced by the monitor is less than 3% for all three measures.

We also compared the cost of using our system calls against the cost of obtaining the same information through the standard facilities (i.e., `/proc`) provided by the Linux operating system. The costs are show in Table 4.

<b>Server Module Measures</b>	
<b>Measure</b>	<b>Description</b>
conn/s	number of connections to the server per second
Mbit/s	number of Megabits served per second
reads/conn	number of read calls per connection
rblk/conn	number of blocks read per connection
p_parsetime	processor time spent parsing the request
p_proctime	processor time spent processing the request
p_logtime	processor time spent performing standard HTTP logging
e_parsetime	elapsed time spent parsing the request
e_proctime	elapsed time spent processing the request
e_logtime	elapsed time spent performing standard HTTP logging
<b>Kernel Module Measures - my_get_kstats</b>	
cpu_user(%)	percentage of elapsed time spent in user mode
cpu_sys(%)	percentage of elapsed time spent in kernel mode
cpu_idle(%)	percentage of elapsed time cpu was idle
reads/s	total number of read operations per second
rblk/s	total number of blocks read per second
writes/s	total number of write operations per second
wblk/s	total number of blocks written per second
pagein/s	number of pages the system paged in (including those found in buffer cache) per second.
pageout/s	number of pages the system paged out (including those found in buffer cache) per second.
interrupts/s	number of interrupts from all devices per second
net_intrpt/s	number of interrupts from network interface per second
disk_intrpt/s	number of interrupts from disk driver per second
context swtch/s	number of context switches per second
<b>Kernel Module Measures - my_get_procstats</b>	
cpu(%)	percentage of cpu used by all copies of the monitored program
mem(%)	percentage of memory used by all copies of the monitored program
maj_flt/s	number of major faults all copies of the monitored program have made per second [22]
started processes	total number of copies of the monitored program
running processes	number of copies of the monitored program waiting for run time.

Table 2: Description of Server and Kernel Module measures.

	<b>With Monitor</b>	<b>Without Monitor</b>	<b>Difference (%)</b>
Conn/s	16.98	17.28	1.74
Mbits/s	3.80	3.90	2.56
Response Time (sec)	1.78	1.74	1.99

Table 3: Server and Kernel Module overhead measurements.

Type of statistic	/proc	syscall
kernel	1610	8
network	1082	17
process	10000	2000

Table 4: Cost (in microseconds) of the Monitor Instrumentation

## 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. There are two noteworthy results from these experiments. First, because of the way WebStone works, we saturated the CPU at the server with more than 5 WebStone clients, no matter how many clients were instantiated. Second, the fraction of memory consumed by the HTTP processes at the server grew linearly with the number of WebStone clients. We found that memory became saturated at the server with 30 clients. Therefore, our results 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 5 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 most interesting result in Table 5 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

	Class 1	Class 2	Class 3
<b>conn/s</b>	16.18	0.77	0.08
<b>Mbits/s</b>	1.52	0.99	1.44
<b>reads/conn</b>	0.03	3.03	38.22
<b>rblk/conn</b>	0.06	6.05	76.43
<b>p_parsetime(ms)/conn</b>	5.03	4.92	3.32
<b>p_proctime(ms)/conn</b>	19.53	147.68	2214.39
<b>p_logtime(ms)/conn</b>	5.62	7.25	6.85
<b>e_parsetime(ms)/conn</b>	26.84	20.06	3.34
<b>e_proctime(ms)/conn</b>	163.66	3734.01	55573.82
<b>e_logtime(ms)/conn</b>	720.46	810.89	905.68

Table 5: Server Module Results for 30 Clients.

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 5 which show the same increase are the read calls per connection and the blocks read 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 5 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 request size), the bandwidth that each class consumes on the network is about the same (i.e., between 1 and 1.5 Mbps). This is due to the heavy-tailed nature of the file size distribution.

The results from Table 5 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 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 Server Module Validation

To validate the SM, we compared its measurements with those made by WebStone. SM counts user space events on the server, which can be matched with user space events on the client (observed by WebStone). Table 6 compares aggregate SM measurements from Table 5 with the same measurements from WebStone. Note that the differences between both Connections and Megabits per second, measured by the client and server, are less than 1%.

	Server Module	WebStone	Difference (%)
Conn/s	17.03	17.02	0.05
Mbits/s	3.95	3.98	0.76

Table 6: Server Module Validation against WebStone.

### 5.3 Kernel Module Results

Table 7 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 7 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 and block reads 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.

	30 clients
cpu_user(%)	8.85
cpu_sys(%)	88.70
cpu_idle(%)	2.45
reads/s	5.77
writes/s	3.96
rblk/s	11.53
wblk/s	7.92
pagein/s	151.31
pageout/s	7.05
interrupt/s	1010.85
net_intrpt/s	594.19
disk_intrpt/s	316.67
context swtch/s	58.13

Table 7: Kernel Module Results for `my_get_kstats`

HTTPD	30 clients
cpu(%)	91.62
mem(%)	99.98
started processes	26.60
running processes	23.39

Table 8: Kernel Module Results for `my_get_procstats`

We have seen a correspondence between SM and KM statistics looking at Table 5 and then Table 7. We also wanted to show a correspondence in the reverse direction. Table 8 shows aggregate process statistics for the HTTP processes, measured in the kernel. Note that the CPU is over 90% utilized, and that memory is almost 100% utilized by the HTTP processes alone. 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 23 processes to service 30 concurrent connections. 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.

#### 5.4 Kernel Module Validation

The validation of the results collected by KM was done through comparisons with similar measurements obtained through the /proc filesystem interface. Table 9 shows the percentages and rates calculated from the information read from /proc and the same values measured by KM. Note that the difference between the same measurements from /proc and KM are less than 1%.

	Results from /proc	Results from KM	Difference (%)
<b>cpu_user(%)</b>	8.21	8.21	0
<b>cpu_sys(%)</b>	84.20	84.74	0.64
<b>cpu_idle(%)</b>	7.04	7.04	0
<b>reads/s</b>	4.44	4.44	0
<b>rblk/s</b>	8.88	8.88	0
<b>writes/s</b>	3.97	3.98	0.25
<b>wblk/s</b>	7.94	7.95	0.13
<b>pagein/s</b>	141.09	141.76	0.47
<b>pageout/s</b>	7.54	7.55	0.13
<b>interrupts/s</b>	960.42	960.50	0.008
<b>net_intrpt/s</b>	563.00	563.04	0.007
<b>disk_intrpt/s</b>	297.42	297.45	0.01

Table 9: Kernel Module Validation against /proc.

#### 5.5 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 [16, 14, 17] 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 10 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 10 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 [14, 16]. 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).

TCP State	TIME_WAIT = 0	TIME_WAIT = 60 sec
ESTABLISHED	5.30	26.62
TIME_WAIT	0	921.17
CLOSE	43.33	34.61

Table 10: Number of Connections in TCP States (KM)

TIME_WAIT = 60sec	Class 1	Class 2	Class 3
conn/s	16.08	0.86	0.06
Mbit/s	1.52	1.15	1.17
p_parsetime(ms)/conn	5.04	5.30	6.16
p_proctime(ms)/conn	20.22	144.57	2224.35
p_logtime(ms)/conn	5.62	6.80	5.31
e_parsetime(ms)/conn	23.64	31.93	64.45
e_proctime(ms)/conn	153.62	3630.36	53897.69
e_logtime(ms)/conn	720.14	778.66	583.31

Table 11: Server Module Results for TIME\_WAIT of 60 with 30 Clients

TIME_WAIT = 0	Class 1	Class 2	Class 3
conn/s	15.77	0.81	0.08
Mbit/s	1.49	1.05	1.43
p_parsetime(ms)/conn	1.72	1.59	1.70
p_proctime(ms)/conn	11.79	84.04	1390.05
p_logtime(ms)/conn	4.58	6.42	7.07
e_parsetime(ms)/conn	18.99	21.08	8.34
e_proctime(ms)/conn	22.40	1990.35	10043.23
e_logtime(ms)/conn	53.57	65.97	50.87

Table 12: Server Module Results for TIME\_WAIT of 0 with 30 Clients

To understand the impact this large number of TIME\_WAIT connections has on server performance, we first looked at results from the SM. Tables 11 and 12 show SM results for 30 clients using a TIME\_WAIT time of 60 seconds and 0, respectively. Note first of all that the throughput (conn/s and Mbit/s) does not change significantly. This is true for our experiments since the total response time seen by WebStone clients (including network delay and client processing) is much greater than the time spent processing a request at the server. Furthermore, the client requests are “flow controlled” since a new request isn’t issued by a client until the previous one is completed. However, the results for latency 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 requests) take longer. This is true in terms of both CPU time and elapsed time. For example, the elapsed time to process a large (i.e., Class 3) request is five times greater for a `TIME_WAIT` time of 60 seconds than for a `TIME_WAIT` time of 0.

Given that the latency of different components of servicing an HTTP request is reduced by decreasing the `TIME_WAIT` time to 0, we wanted to understand why. We first examined the resources consumed by the HTTP processes. Table 13 shows these results. Note that the consumption of all resources that we monitor is decreased. Specifically, there are a factor of 7 fewer HTTP processes running to serve the same number of clients. In addition, both CPU and memory utilization are reduced significantly. Table 14 shows system-wide statistics for CPU and

<b>HTTPD</b>	<b>TIME_WAIT = 0</b>	<b>TIME_WAIT = 60 sec</b>
<b>cpu(%)</b>	49.49	87.33
<b>mem(%)</b>	37.79	94.53
<b>started processes</b>	10.31	25.30
<b>running processes</b>	2.95	21.95

Table 13: Kernel Module Results for HTTP processes for different `TIME_WAIT` values

memory utilization. These results are consistent with those in Table 13, and also show that the reduced CPU utilization is due to a reduction in system time (i.e., time spent in the kernel). It is interesting that this is true even though context switching occurs more frequently. Note also that our measures of throughput (e.g., reads/s) remain roughly the same, which is consistent with the SM results in Tables 11 and 12.

	<b>TIME_WAIT = 0</b>	<b>TIME_WAIT = 60 sec</b>
<b>cpu_user(%)</b>	8.733	8.58
<b>cpu_sys(%)</b>	45.622	84.45
<b>cpu_idle(%)</b>	45.643	6.97
<b>reads/s</b>	5.09	4.91
<b>writes/s</b>	3.51	4.07
<b>rblk/s</b>	11.55	9.81
<b>wblk/s</b>	7.30	8.14
<b>pagein/s</b>	121.98	128.55
<b>pageout/s</b>	6.16	7.03
<b>interrupt/s</b>	1004.79	939.33
<b>net_intrpt/s</b>	647.23	568.19
<b>disk_intrpt/s</b>	257.56	271.14
<b>context swtch/s</b>	142.60	60.64

Table 14: Kernel Module Results for different `TIME_WAIT` values

These results clearly show that the impact of the long `TIME_WAIT` holding time is twofold. First, that more HTTP processes are active at the same time. Second, that these processes consume more memory and CPU time (and system time in particular) to serve the same number of clients. However, we are currently unable to identify the main cause of the increase in CPU time or memory consumption. It is clear that increased pressure in the

memory system (i.e., caches, TLB's, page tables, etc.) would increase the time to process an HTTP request, however, we don't know by how much. Answering this question is the main focus of our future work. 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 3%). 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, over 90% of the time spent handling HTTP requests is spent in the kernel. Furthermore, keeping TCP connections open, as required by TCP, causes a factor of 2-9 increase in the elapsed time required to service an HTTP request. Data gathered from Webmonitor provide insight into the causes of this performance penalty. Specifically, we observe a significant increase in resource consumption along three dimensions: the number of HTTP processes running at the same time, CPU utilization, and memory utilization.

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.

## References

- [1] V. Almeida, A. Bestavros, M. Crovella, and A. Oliveira. Characterizing reference locality in the WWW. *Proceedings of IEEE-ACM PDIS'96*, December 1996.
- [2] P. Aoki, A. Woodruff, E. Brewer, P. Gauthier, and L. Rowe. An investigation of documents for the world wide web. *Proc. of the Fifth World Wide Web Conference*, May 1996.
- [3] M. Arlitt and C. Williamson. Web server workload characterization. *Proc. of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.
- [4] M. F. Arlitt. A performance study of Internet web servers. M.Sc. Thesis, Department of Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, June 1996.

- [5] K. Birman and R. van Renesse. Software for reliable networks. *Scientific American*, May 1996.
- [6] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo I. Seltzer, and Michael D. Smith. The measured performance of personal computer operating systems. *ACM Transactions on Computer Systems*, 14(1):3–40, February 1996.
- [7] A. Cockcroft. Watching your web server. *SunWorld Online*, March 1996. URL: <http://www.sun.com/sunworldonline/swol-03-1996/>.
- [8] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *Proc. of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.
- [9] G. Serazzi D. Ferrari and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice Hall, Englewood Cliffs, 1983.
- [10] Simson L. Garfinkel. The wizard of Netscape. *WebServer Magazine*, 1(2):59–63, 1996.
- [11] Nicholas Gloy, Cliff Young, J. Bradley Chen, and Michael D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proc. of 23rd International Symposium on Computer Architecture*. ACM/IEEE, May 1996.
- [12] K. Lai and M. Baker. A performance comparison of UNIX operating systems on the Pentium. In *Proceedings of the 1996 USENIX Conference*, San Diego, CA, January 1996. USENIX.
- [13] D. Menasce, V. Almeida, and L. Dowdy. *Capacity Planning and Performance Modeling*. Prentice Hall, Englewood Cliffs, 1994.
- [14] Jeffery C. Mogul. Network behavior of a busy Web server and its clients. Research Report 95/5, DEC Western Research Laboratory, October 1995.
- [15] Jeffery C. Mogul. Operating system support for busy Internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.
- [16] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 299–313, Cambridge, MA, August 1995. ACM.
- [17] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. In *Proceedings of Second WWW Conference '94: Mosaic and the Web*, pages 995–1005, Chicago, IL, October 1994.
- [18] D. Robinson and the Apache Group. *APACHE - An HTTP Server, Reference Manual*, 1995. URL: <http://www.apache.org>.
- [19] Y. Somin S. Agrawal, M. Forsyth. Measurement and analysis of process & workload CPU times in UNIX environments. *Proceedings of the CMG'96*, December 1996.
- [20] R. McGrath T. Kwan and D. Reed. NCSA's world wide web server: Design and performance. *IEEE Computer*, November 1995.
- [21] G. Trent and M. Sake. *WebStone: The First Generation in HTTP Server Benchmarking*, February 1995. URL: <http://www.sgi.com/Products/WebFORCE/WebStone/paper.html>.
- [22] M. Welsh. *The Linux Bible*. Yggdrasil Computing Incorporated, 2<sup>nd</sup> edition, 1994.