SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Understanding Web server configuration issues

Martin Arlitt*,† and Carey Williamson

Department of Computer Science, University of Calgary

SUMMARY

This paper proposes a methodological approach to the evaluation of Web server performance in a simple LAN test environment. The paper examines how different system and application configuration parameters can, over a range of workloads, impact the performance of a Web server. Our approach relies on relatively fine-grain reporting of performance data for a broad set of system-level metrics. Graphical visualization of these performance indices helps to identify the primary system bottleneck in each configuration studied.

The Apache Web server is used as a case study to demonstrate the methodology. Our experiments quantify the performance implications of several configuration decisions common to any Web server implementation, and also serve to illustrate several performance anomalies specific to the Apache Web server (if misconfigured).

KEY WORDS: Web servers; Web performance; benchmarking; validation; performance methodology

1. INTRODUCTION

Web server performance plays a central role in the user-perceived "Internet experience", and thus has been the focus of a lot of recent research [3, 4, 5, 12, 18, 19, 25]. Many different Web server architectures have been proposed, and evaluated on a wide range of platforms [5, 12, 18, 19, 22, 27]. Many implementation optimizations have been proposed for Web servers as well, particularly in regards to communication protocol handling [3, 4], request scheduling [7], and reducing system overhead [3, 18].

One of the challenges in comparing and evaluating Web servers is the sensitivity of Web server performance to a broad set of system-level and application-level configuration parameters. For example, some Web servers are process-based, some are not; some Web servers

*Correspondence to: Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, AB, CANADA T2N 1N4
†E-mail: {arlitt,carey}@cpsc.ucalgary.ca

Received August 2002

use memory-mapped files, some do not; some Web servers are highly optimized for static content retrieval, while others are not. Understanding the performance tradeoffs between these configuration choices is complicated. In addition, the characteristics of the workload presented to the Web server can have a dramatic effect on the observed performance. In fact, fundamentally different server behaviours can result for different workloads, depending on whether the primary system bottleneck is the CPU, the network bandwidth, or the network latency [19].

Understanding the performance implications of different Web server configurations is particularly relevant in the validation stage of any performance evaluation study. The validation process entails verifying the accuracy of measurements, explaining any counter-intuitive results, and investigating anomalous or unexplained behaviours [16]. Furthermore, the results obtained from the study must be repeatable: re-running the experiments on the same system or in a similar test environment (whether by the original researchers or by others) must yield equivalent results.

This paper describes work that was done as part of the validation process for our work on Web server benchmarking using parallel WAN emulation [25]. For simplicity, we focus only on the LAN environment in this paper. The paper explores the implications of different system and application configurations on the performance of a Web server. Our approach relies on relatively fine-grain reporting of performance data for a broad set of system-level performance metrics. Graphical visualization of these performance indices helps to identify the primary system bottleneck in each configuration studied. The visualization process makes results more understandable, building the intuition required for the identification of bottlenecks in new test configurations as they are explored. Data visualization also makes performance anomalies immediately apparent, when they occur. Our intent is not to instruct all administrators how to configure their servers; rather, we aim to provide a better understanding of the potential impacts of different configuration decisions. In addition, we demonstrate how others can evaluate the impact of various workload and configuration factors in a systematic fashion.

The main contributions of the paper are as follows. First, we describe a methodological approach to Web server benchmarking in a simple LAN test environment, codifying certain common-sense principles followed by many other researchers. Second, we show that (as expected) some system-level and application-level parameter settings can have a significant impact on the performance of the Web server. For example, the use of persistent connections and pipelining can significantly improve performance, while unnecessary process creation and termination can degrade performance. Third, we demonstrate that some components of a system behave in an unexpected manner when a bottleneck is impeding the system. As a result, the methodology highlights the (typical) need to examine the behaviour of multiple components in order to correctly identify the system bottleneck. Through our efforts, we have gained a better understanding of how Web servers behave in the presence of different bottlenecks. This knowledge provides useful insight into the validation and understanding of results from more elaborate Web server benchmarking studies. Our methodology is also general enough to apply to a wide range of performance evaluation studies beyond Web server benchmarking.

There are several intended audiences for this paper. First, performance analysts may be interested in our instrumentation and graphical visualization approaches to identifying system bottlenecks. Second, researchers familiar with Web server benchmarking can learn about the process behind properly tuning the Web server in our benchmarking study. Finally, system/network administrators can gain insight into the performance implications of different configurations on their operational Web servers.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the test environment used in this study. Section 4 presents our methodology and experimental design, while Section 5 presents the results of our study. Section 6 concludes the paper with a summary of our work and a discussion of future directions.

2. RELATED WORK

The work presented in this paper was initiated while evaluating the effects of Wide-Area Network (WAN) characteristics on Web server performance [25]. As part of the validation process for that study, we needed to first understand the behaviour of a Web server in a simple Local Area Network (LAN) environment. In other words, we needed to distinguish between Web server behaviours that were caused by WAN characteristics and those that were simply due to the configuration of the Web server.

There are several research studies that have evaluated the performance of different Web servers (e.g., [12, 13, 18]). There are also informal guidelines for tuning the Apache Web server [1, 9]. While such studies and guidelines are complementary to our own, they typically do not provide detailed information on the performance implications of specific system and application configurations used. In addition, we elaborate on our methodology, to enable others to conduct their own experiments in a consistent manner.

There have also been many commercial Web server benchmark studies. For example, many server companies use standard benchmarks such as SPECWeb [26] to provide a measure of the performance of their products relative to those of their competitors. When publishing SPECWeb results a company must disclose a detailed list of the system and application configurations. Many of these disclosures contain dozens of configuration changes. However, little or no information is provided regarding the performance impacts of these changes. In this paper, we focus on evaluating the performance impact of a small set of parameters on the Apache Web server. We examine Apache since it is the most commonly used Web server on the Internet today [20], primarily because it is feature-rich and freely-available. Most server companies provide SPECWeb results for only less popular, but more performant Web servers.

Benchmarking of Web caching products has received significant attention in recent years. Since Web proxy cache workloads vary significantly from Web server workloads, alternative benchmarking tools are required. One of the most complete tools for benchmarking such caches is Web Polygraph [24]. This tool can not only generate realistic Web cache workloads, but also evaluates the performance based on metrics of specific importance for caching, including measures of hit ratios and object freshness.

Graphical visualization of performance data is common practice in many companies and organizations, and is often crucial to identifying interesting phenomena. For example, Barford and Plonka [6] report how the University of Wisconsin uses this approach to detect anomalies in network traffic behaviour. In many cases, the key to identifying each type of anomaly is

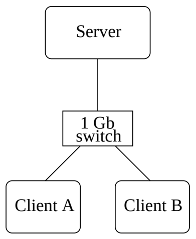


Figure 1. Simple LAN test environment for Web server experiments

to combine information from multiple sources [6]. In this paper, we similarly demonstrate the importance of monitoring multiple components in order to identify the system bottleneck correctly.

3. EXPERIMENTAL ENVIRONMENT

This section describes the simple LAN test environment used for the experiments in this paper. The testbed, as shown in Figure 1, consists of two clients that submit requests to a Web server over a 1 Gb/s full-duplex switched-Ethernet LAN. The remainder of this section provides a detailed description of the test environment.

3.1. Equipment and configuration

3.1.1. Client configuration

The client machines in our testbed are IBM x335 servers running RedHat Linux 8.0. Each client machine has a single Intel 2.4 GHz Xeon processor, 1 GB RAM, a 36 GB 15K SCSI disk, and two 1 Gb/s Ethernet NICs (although only one NIC in each machine is used in our experiments). Each client uses 25 MB of memory as a "RAMdisk" (i.e., a virtual disk [21]) for collecting statistics on the client behaviour during testing. Using a RAMdisk eliminates writing to the physical disk during testing.

The Linux `/proc` directory [23] is used to monitor the behaviour of the system (see Section 3.2.3) and to communicate configuration changes to the kernel. In particular, we used this interface to make several modifications that allow the clients to generate and sustain very high connection request rates, even when the server is overloaded. First, we increased the number of available file descriptors (`/proc/sys/fs/file-max`) to 32,768. We also enabled TCP TIME_WAIT recycling (`/proc/sys/net/ipv4/tcp_tw_recycle`) to free up sockets in a TIME_WAIT state more quickly. Next, we increased the size of the local port range (`/proc/sys/net/ipv4/ip_local_port_range`) that is available for use. Finally, since we were not running the workload generators as root, we had to modify the PAM

(Pluggable Authentication Module) configuration to allow us to utilize all 32,768 file descriptors rather than the default of 1,024. This requires modifying `/etc/security/limits.conf` and `/etc/pam.d/ssh`.

All non-essential processes on the client machines were disabled, to minimize the consumption of resources by processes unrelated to workload generation. This step removes "noise" from the performance data collected, allowing us to identify more readily the impacts of specific configuration changes.

3.1.2. Server configuration

The Web server platform is an IBM x335 server running RedHat Linux 7.3. The server has a single Intel 2.4 GHz Xeon processor, 1 GB RAM, a 36 GB 15K SCSI disk, and two 1 Gb/s Ethernet NICs (only one of which is used in this study). A 25 MB RAMdisk is used for storing system measures during testing.

As with the client machines, all non-essential processes on the server were disabled prior to conducting tests, to minimize consumption of system resources. We also modified the server's kernel configuration, increasing the number of available file descriptors to 32,768.

3.2. Software configuration

3.2.1. Client workload generation

The primary workload generation tool in our test environment is `httperf` [17], a tool for measuring HTTP performance. We chose to use this tool for several reasons. First, we have used this tool in the past, so we are familiar with its interface and its capabilities. Second, `httperf` supports a wide-range of features (e.g., persistent connections, pipelining, SSL) that are useful for testing Web server functionality. Third, `httperf` is available [11] in source code form, so that we can easily add desired functionality (e.g., to print server response rate information more frequently).

A second workload generation tool used is `netperf` [15]. The `netperf` utility is a network benchmark that focuses primarily on bulk data transfer and request-response performance using either TCP or UDP via the BSD socket interface [14]. We use `netperf` for validating network performance results.

3.2.2. Web server software

The majority of our experiments used the Apache Web server [2] (version 1.3.28), a process-based server that uses a separate process to handle each outstanding request. We chose to use Apache for several reasons. First, Apache is currently the most common Web server on the Internet, used by approximately 60% of all Web sites [20]. Second, Apache has been ported to many platforms, including Linux (used in our LAN test environment) and Tru64 (used in our WAN emulation environment). Third, the source code for Apache is available, which allows us to learn more about its operation, and to make modifications to the server if needed.

The purpose of this paper is to understand the implications of different configurations on the performance of a Web server; it is *not* our goal to look for performance problems specific to Apache. We note that, as stated on the Apache Web site, Apache is designed to be "correct first, and fast second". We simply use Apache, for the reasons stated above, to demonstrate the performance impacts of Web server configurations (and misconfigurations). Clearly, it is possible to misconfigure any server.

The other Web server that we use is Tux [27] (version 2.1). Tux is a kernel-based, multi-threaded, high performance HTTP server available for Linux systems. We use Tux to verify that our client workload generators are not the bottleneck in any of our tests.

3.2.3. Monitoring software

Critical to our work is the availability of timely and detailed data on the state of the system under study. We rely on several sources for this information. Much of our data is acquired using the Linux utility `sar` (system activity report) [10]. Using `sar`, we are able to monitor a wide range of system activities (e.g., CPU utilization, I/O transactions). Statistics on the Linux networking subsystem are obtained using the `netstat` command. In particular, `netstat` can provide information on network-related errors (e.g., dropped TCP connections). This information is collected before and after every test. Information on TCP and HTTP behaviours is available from the output of `httperf`. This output provides the response rate on a second-by-second basis, and other statistics (e.g., average TCP connection rate, HTTP request rate, and HTTP reply rate) on a test-by-test basis. Additional information is collected from a variety of log files. The Apache error log reports warnings and errors specific to the Apache server. In addition, `syslogd` can be used to record messages on system activity, and to trap kernel messages. The latter log is useful for ensuring that the kernel is functioning properly. These logs are examined after the conclusion of an experiment.

3.3. Controlling the test environment

In this paper, we define an *experiment* as a number of *tests*, each of which examines a different *level* of a particular *factor*. All other factors are fixed throughout the experiment, although they can vary between experiments.

Each experiment is controlled from one of the client machines (Client A). Each experiment is specified as a shell script, which is then executed on the control machine. Controlling the experiments in this way ensures that the tests are conducted in a consistent manner. Archiving the scripts aids in repeating the results as well.

Prior to the start of each experiment, the control machine communicates (via `ssh`) with each machine involved in the experiment. Prior to starting the initial test, information on the current state of each machine is collected. The control machine then starts the monitoring software on all test systems. The control machine is also used to start each test, and to collect data after tests complete. At the completion of the experiment, all of the collected data is archived to disk for off-line analysis.

4. PERFORMANCE EVALUATION METHODOLOGY

4.1. Principles

Our performance evaluation study follows several key principles, which we briefly describe here. While most of these principles are common-sense and not highly original, we think it is important to state them explicitly, thus codifying them as part of our methodology, and justifying some of the design choices made in our study.

The key principles are:

- **Keep it Short and Simple** (KISS). The KISS principle has been around for many years, and applies in many different contexts, including software design and program debugging. In simple terms, it means to include everything that is crucial to your purpose, while excluding everything that is not. We apply this principle by choosing a simple LAN test environment for our study (the WAN test environment easily merits another paper). We also use this principle to justify a simple two-client test network, with each client retrieving fixed-size static content from the Web server.
- **Instrumentation Necessary for Knowledge** (INK). Adequate instrumentation is crucial to the characterization of system performance and the identification of performance anomalies. In our study, we use relatively fine-grain (i.e., five second intervals) reporting of performance data. Past experience suggests that aggregating performance data over longer time scales tends to miss short-time-scale performance anomalies.
- **Multiple Observational Viewpoints** (MOV). When identifying performance bottlenecks or anomalies, it is important to have measurements from multiple perspectives [6]. For example, Barford and Crovella [5] recommend simultaneous collection of client, server, and network performance data in order to quantify wide-area Web performance. We apply a similar principle to our study so that we can separate client-side and server-side effects from network-level effects.

4.2. Overview of methodology

When conducting a performance evaluation study, identifying the bottleneck resource can be just as important as (or even more important than) arriving at a set of numbers that quantify the performance of the system. When evaluating a complex distributed system, correctly identifying the bottleneck resource can be quite challenging. Furthermore, when one bottleneck is alleviated, a new one appears elsewhere in the system, and the search must begin anew. When the cost of removing the bottleneck exceeds the benefit of doing so, the exercise ends.

There are numerous problems in identifying bottlenecks. For example, if the system under study is proprietary, it may be difficult or impossible to distinguish between many potentially limiting resources. Similarly, if measurement data is not available in a timely fashion, multiple resources may behave erratically, making it difficult to identify the true bottleneck. If measurement data is not available for all resources in the system, finding the bottleneck resource can be difficult.

Table I. Experimental workload factors and levels (default values in **bold**)

Factor	Description	Levels	Section
File Size	Web document transfer size requested	**1 KB**, 1 MB	5.1,5.2
Connection Rate (1 KB)	TCP connection rate (1 KB transfers)	500-10,000/second	5.1
Connection Rate (1 MB)	TCP connection rate (1 MB transfers)	10-200/second	5.2
Persistent Connection (1 KB)	requests/connection (1 KB transfers)	**1**,2,4,8	5.1.6
Pipelining (1 KB)	pipelined requests (1 KB transfers)	**1**,2,4,8	5.1.6

Table II. Experimental Apache server configuration factors and levels (default values in **bold**)

Factor	Description	Levels	Section
access log	record access log of all client requests	**disabled**, `enabled`	5.1.2
`MaxRequestsPerChild`	maximum requests per server process	**0**, 1,000, 10,000, 100,000	5.1.3
`MMapFile`	map file into server memory at startup	**disabled**, enabled	5.1.4
`MaxClients`	maximum simultaneous requests in service	128, **150**, 256, 512, 1024	5.1.5
`MaxSpareServers`	maximum idle server processes	**10**, 128, 256, 512, 1024	5.1.5
`MinSpareServers`	minimum idle server processes	**5**, 128, 256, 512, 1024	5.1.5
`StartServers`	initial number of server processes	**5**, 128, 256, 512, 1024	5.1.5

We are fortunate to have timely access to a lot of measurement data in our test environment. Furthermore, we were able to obtain additional data by modifying the open source code that we were using. In addition to learning about the performance implications of various configuration parameters, we utilized the available data to recognize how a system behaves in the presence of various bottlenecks.

Our approach to understanding the behaviour of the system is as follows. We begin by examining the system in an idle state (i.e., when the server is not receiving or servicing any requests). This step provides us with the information needed to filter out activity unrelated to the Web server testing. We then conduct tests in a controlled manner, purposely keeping the tests as simple as possible. While this might not exercise the Web server in the most realistic manner, we believe that our approach is sufficiently realistic to enable us to identify correctly the causes of various behaviours. As a better understanding of system behaviours is acquired, the degree of realism in the experiments can be increased. If the initial experimental design is too complex, it may be difficult to ascertain the true cause of certain behaviours.

4.3. Experimental design

We explore eleven factors in our experiments using a one-factor-at-a-time experimental design [16]. The factors include four workload factors and seven application configuration parameters. Table I summarizes the workload factors and levels used in our experiments, while Table II lists the Apache server configuration factors and levels. The default values used for each factor are shown in bold.

The workload factors allow us to exercise the server in a variety of ways. Of particular importance in our experiments is the duration that TCP connections are held open. Since the Apache 1.3.28 server uses a process-based model, the number of Apache processes limits the number of concurrent TCP connections that can be serviced simultaneously. Since our experiments are conducted in a LAN environment, and all our requests are for static files, the duration of each experiment is (typically) related directly to the file size requested. Thus we focus on tests involving two file sizes: small (1 KB = 1,024 bytes) and large (1 MB = 1,048,576 bytes). The lifetime of the TCP connections used to exchange a single request for and response of a small file is dominated by the time to perform the TCP handshakes to establish and terminate a connection. These experiments stress the server's ability to accept new requests. We also examine the performance impact of re-using TCP connections, both with and without pipelining of requests for 1 KB files. The TCP connection lifetime for the large files is dominated by the time to transfer the files. These tests stress the server's ability to send data to the client.

The Apache server configuration factors allow us to influence the Web server's behaviour in the following ways. The `MMapFile`, and logging factors affect the amount of disk traffic that occurs. The `MaxClients`, `MaxRequestsPerChild`, `MinSpareServers`, `MaxSpareServers` and `StartServers` parameters affect the creation and termination of Apache server processes.

4.4. Validation of test environment

In this section, we provide a basic "sanity check" of our experimental environment and its instrumentation. We first study the idle test system, and discuss the steps taken to make the system as "clean" as possible; next, we verify the correct operation of the client workload generators.

4.4.1. The idle system

When we initially monitored our test system (with a default Apache/Linux configuration) during an idle period (i.e., no test in progress), we were surprised to find a non-trivial amount of activity occurring. This activity included packets arriving on the network interfaces, files being read and written, and several processes consuming varying amounts of CPU time. After some effort, we were able to identify the causes of all of the extraneous behaviours. We then "cleaned-up" the system by disabling system processes not required for our testing.

4.4.2. Client workload generation capabilities

The next step in our study was to verify that the clients could generate the targeted range of connection and request rates. That is, we want to demonstrate that our clients can generate and sustain a rate of (at least) 10,000 requests per second for a static 1 KB file. We also need to confirm that the network, server hardware, and monitoring infrastructure can handle (at least) 10,000 requests per second. For the validation experiments only, we use the Tux Web server. All remaining experiments use the Apache server.

In the 1 KB validation experiments, the clients start by generating 500 requests per second, for a sustained two-minute interval. The clients then increment the request rate by 500 requests per second, and use that target rate for the next two-minute interval (with a slight pause between rate changes). The clients continue in this fashion up to a peak connection rate of 20,000 per second is reached.

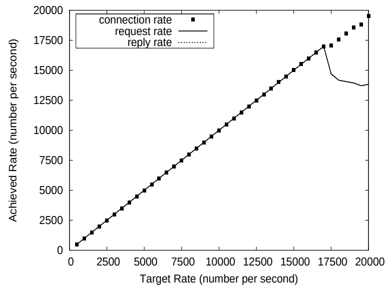


Figure 2. Connection, request and reply rates for 1 KB workload, Tux server

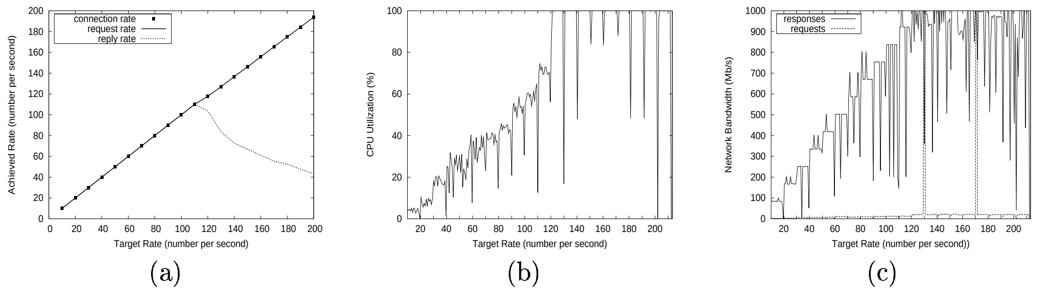


Figure 3. Validation Information for 1 MB workload, Tux server

Figure 2 shows the results from the 1 KB workload validation tests. In this graph there are three sets of data plotted. First, the points (black squares) represent the average number of TCP connections per second initiated by the clients in the two-minute test. Second, the solid line (which happens to overlay the points throughout most of the graph) shows the average rate at which HTTP requests are issued to the server. Third, the dotted line (also overlaid on most of the points in this plot) shows the average number of HTTP responses received per second from the server in each two-minute test. Graphs of this form are used throughout the paper to illustrate performance results.

Figure 2 shows that the clients were able to generate and sustain rates of up to 20,000 TCP connections per second, and 17,000 HTTP requests per second. In addition, these results indicate that the server hardware can issue 1 KB responses at a rate of (at least) 17,000 responses per second. Since we do not anticipate the Apache server outperforming Tux, we do not attempt to alter the configuration of the Tux server to further improve its performance. Should Apache achieve lower performance than that of Tux, it indicates a bottleneck related to the Apache server software, and not to our testing infrastructure.

We also use the Tux Web server to examine the ability of our test environment to support the transfer of 1 MB files. Figure 3 shows the results. Figure 3(a) indicates that our clients are able to establish up to 200 TCP connections per second, and issue up to 200 HTTP requests per second. However, the server can only send a maximum of 110 1 MB responses per second. As the request rate increases beyond this point, the response rate continually decreases. Figure 3(b) shows that the server CPU utilization is 100% for request rates above 110 per second. However, this is not the bottleneck in the system. Figure 3(c) reveals that the network link between the Web server and the Ethernet switch is saturated, with average utilization above 900 Mb/s, and peak utilization reaching 1 Gb/s, once the request rate exceeds 110 per second. This is an example of why it is important to monitor multiple components in order to properly identify the system bottleneck. Although the server CPU utilization reaches 100%, this is a result of the network bottleneck. To identify the true bottleneck, it is important to determine the order in which events occur; fine-grain instrumentation helps in this case.

These experiments establish confidence in the suitability of our simple experimental infrastructure. The experiments in the next section use this infrastructure to assess the performance implications of different Web server configurations.

5. EXPERIMENTAL RESULTS

In this section, we present the results of experiments involving different Web server configurations. Section 5.1 describes the experiments that used 1 KB transfers. Experiments conducted with 1 MB transfers are discussed in Section 5.2. Section 5.3 summarizes our results.

5.1. Experiments utilizing 1 KB workloads

This section presents the results of experiments involving 1 KB workloads.

5.1.1. Baseline Apache performance (default configuration)

In this experiment, we study the performance of the Apache Web server using its default configuration. This experiment has access logging turned off, and memory mapping is not used. The Apache defaults for `StartServers`, `MinSpareServers`, `MaxSpareServers`, `MaxClients`, and `MaxRequestsPerChild` are used (5, 5, 10, 150 and 0 (unlimited), respectively). Figure 4 shows the performance results from this experiment.

Figure 4(a) provides the performance results for the default Apache server configuration in our test environment. Two distinct phases of operation can be seen in this graph. On the lower left hand side, the server is operating normally, and is able to accept and respond to all client requests. Above the peak achieved rate of 5,700 per second, the achieved request and response rates begin to deviate from the target rates. This occurs because the server is now in an overloaded state.

From Figure 4(a) we can tell that the server platform operates correctly throughout the experiment, as it continues to establish TCP connections up to the maximum rate of 10,000 per second. Furthermore, we established in Section 4.4.2 that our infrastructure is capable

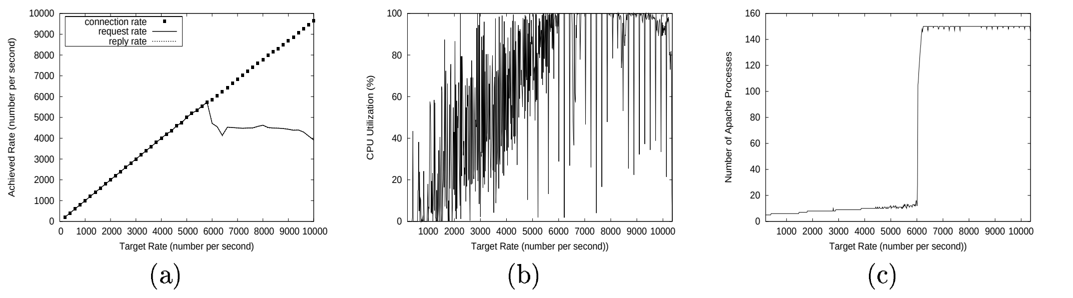


Figure 4. Performance results for the baseline (default) Apache configuration

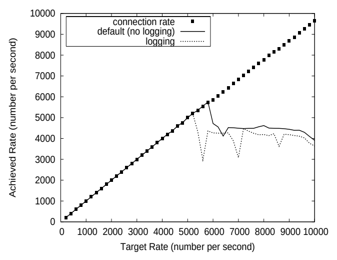
of handling more than 10,000 requests and responses per second. Thus, we know that the bottleneck is related to the Apache server application.

Figure 4(a) indicates that once the server enters the overloaded state, the achieved request rate is well below the target rate. At first glance, the bottleneck might seem to be that the clients are not capable of generating the target request rate, and that the server is simply responding to all of the requests that it receives. However, we know that this is not the case, as we have already demonstrated the workload generation capabilities of the clients far exceed this rate. What is actually occurring in the overloaded state is that the server is too busy to accept all of the incoming requests. As a result, the server actually limits the achieved request rate, and thus controls the response rate as well. This is another example of why it is important to evaluate the system from multiple perspectives, to avoid misidentifying the bottleneck. Although we used a separate experiment to demonstrate that the clients are not the bottleneck, the same conclusions could be reached by analyzing the `httperf` and `netstat` outputs for this experiment.

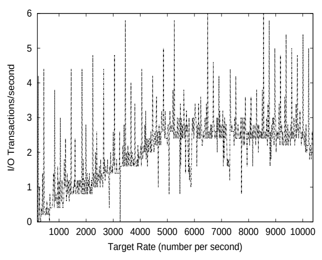
Figures 4(b) and (c) provide insights into what is occurring on the server. Figure 4(b) shows that the CPU utilization (as expected) increases as the issued request rate increases. After request rates above 5,700 per second, the CPU utilization has reached 100%. Figure 4(c) shows that until the bottleneck occurs, Apache is able to satisfy most requests quite quickly, and as a result, only needs to spawn a few new worker processes per test. Once the bottleneck occurs, Apache quickly tries to generate new processes, in order to accommodate new requests, as the existing processes are busy servicing outstanding requests. Under this overload condition, the number of Apache processes quickly increases to the maximum of 150, specified by `MaxClients`.

5.1.2. Logging client requests

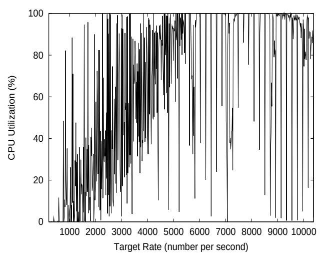
An oft-mentioned approach for improving the performance of a Web server is to disable the logging of client requests (which we have already done in our previous experiment). However, logging is important in numerous circumstances. In this section, we quantify the penalty incurred by enabling the Apache `CustomLog`.



(a)



(b)



(c)

Figure 5. Performance Results with Logging

Figure 5(a) compares the response rates that were achieved with and without logging of client requests. With logging enabled, the server supported approximately 5,200 requests per second. This represents a 10% decrease in performance, compared to the same system with no logging of client requests.

Figure 5(b) shows the I/O activity on the Web server during the logging experiment. These results show that enabling logging results in relatively few I/O transactions per second. However, Figure 5(c) shows that enabling logging results in slightly higher CPU utilization (compared to Figure 4(b)), resulting in a CPU bottleneck at a lower rate than in the experiment without logging.

We believe our results in this section are optimistic, for several reasons. First, we are only recording 74 bytes per request. In operational servers, more data is often recorded (e.g., queries, cookies, referring URLs, user-agent). Recording more data will likely slow the server further. In addition, the server in our test environment has a high performance disk; we expect that servers equipped with slower disks will experience a greater performance penalty for logging client requests. The safest interpretation of our results is that logging reduces server performance by *at least* 10%.

For all remaining experiments, we leave the logging of client requests disabled.

5.1.3. Maximum requests per child process

On some platforms, memory leaks are known to wreak havoc with the performance of the Apache Web server. Because of this, the Apache server provides a simple mechanism to mitigate these problems. This is done through the `MaxRequestsPerChild` server directive. In most cases this parameter should not be changed from the default of 0, which allows each process to handle an unlimited number of requests. Setting this parameter to a different value will cause each child process to terminate after serving the specified number of requests, freeing up its memory resources for use by newly created processes.

Figure 6 shows the performance results when each of the 150 child processes is limited to serving a particular number of requests. The results in Figure 6 show that the performance can

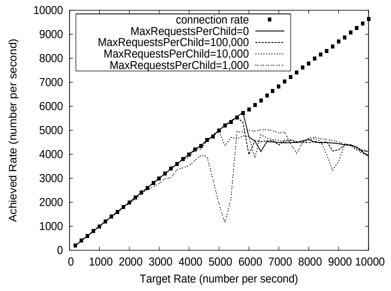


Figure 6. Effect of number of requests per child on Apache performance

suffer substantially if the limit is too restrictive. The default Apache configuration does not place a limit on the number of requests that a process can serve. The maximum performance, as we have already discussed, is 5,700 responses per second. If an overly restrictive limit is used (e.g., 1,000 requests per child process), the performance of the server can be cut in half (or worse). The Apache configuration file suggests a value of 10,000, if the operator is concerned about memory leaks. With this value, the server performance is reduced by approximately 15%.

The drop in performance that occurs when using this mechanism is due to the high cost of creating a new child process. Under normal conditions, Apache pre-forks a number of child processes. When a client request arrives at the server, an idle child process is ready to respond to it. If there are no idle child processes (i.e., the server is busy, or the children have terminated after serving their quota of requests), a new child process must be created to serve the client's request. Process creation is typically much more expensive than serving the request, thus consuming many extra CPU cycles. In other words, when each child process can only serve a finite number of requests, the server loses the benefits of pre-forking processes, and resembles a forking server.

Our results illustrate why the default `MaxRequestsPerChild` parameter value of 0 (unlimited) should not be changed unless absolutely necessary. Limiting the number of requests that can be handled by each child process seriously degrades the performance of the server. For the remainder of our experiments we use the default value of 0 for this parameter.

5.1.4. Memory-mapped files

In this test scenario, the single 1 KB file requested by the clients is mapped into the server's address space when the server is started. Figure 7 indicates that this has a rather significant impact on the performance of the server, as it is now capable of serving 7,300 requests per second (a 28% improvement in performance) over the default case. Note, however, that even with this improvement, the Apache server remains the bottleneck in the system. For the remaining experiments in this section, we leave memory mapping enabled.

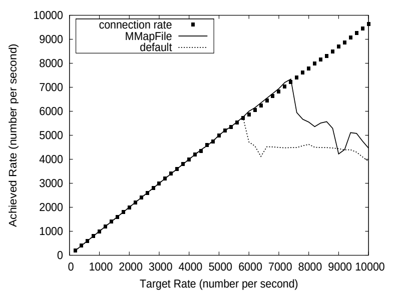


Figure 7. Performance results for memory-mapped files

5.1.5. Server processes

In Section 5.1.1, we observed that the maximum request rate supported by our default Apache configuration was 5,700 requests per second. This rate increased to 7,300 requests per second with memory-mapped files. In both cases, the maximum number of server processes (150) was reached. In this experiment, we compare the default configuration to a new configuration that increases the maximum allowable number of server processes to 1,024. This setting is controlled via the `MaxClients` parameter. In other words, in this experiment we compare the effect of changing `MaxClients` from 150 to 1,024. In order to set `MaxClients` to such a large value, it was necessary to first modify the Apache source code (change `HARD_SERVER_LIMIT` to 1024 in `apache_1.3.28/src/include/httpd.h`) and recompile the server.

Figures 8(a) shows that increasing the maximum number of server processes does not alleviate the bottleneck, as both settings result in the same peak performance. However, using 1,024 server processes results in lower, albeit more predictable performance under overload. Figures 8(b) and 8(c) provide insights into this behaviour. Figure 8(b) reveals that under normal operating conditions, the same number of server processes are used by both server configurations. Once the CPU resources are exhausted, both server configurations attempt to spawn additional server processes to handle the extra client workload. Once 150 server processes have been spawned, the default configuration cannot create any additional processes. Once this occurs, incoming client requests may be dropped. Client requests may also be dropped in the 1,024 server process configuration, as Apache limits the rate at which new server processes can be spawned. However, Figure 8(c) indicates that fewer client requests are being dropped by the 1,024 process configuration (indicated by the number of established connections being dropped from the server's listen queue). Once 150 server processes exist in the default configuration, the server spends no time creating new processes, and less time accepting new requests, since more client requests are being dropped from the listen queue. As a result, the default configuration achieves slightly higher performance under overload conditions. Since the 1,024 process configuration is able to accept and process a larger number

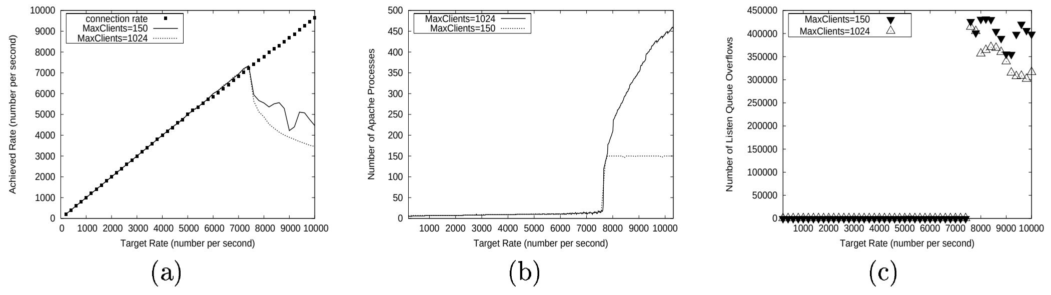


Figure 8. Effect of number of server processes on Apache performance

of requests concurrently, its behaviour under overload is smoother than that of the default server configuration.

These results indicate that limiting Apache to a smaller number of server processes has some benefit under overload conditions, at least in a LAN environment with short-lived conditions. However, having more predictable behaviour under overload may be preferable, especially in situations where connections are long-lived (e.g., WAN environments [3, 19, 25]).

The Apache Web server dynamically creates more server processes (up to a total of `MaxClients` processes) when the currently available processes cannot handle all of the incoming client requests. When the workload subsides, Apache has a similar mechanism to terminate some of the idle processes. One suggestion in Apache performance tuning guides is to disable this mechanism, to avoid the overhead of creating and terminating new processes. To revert to a static number of server processes, one can simply set the value of `StartServers`, `MinSpareServers`, `MaxSpareServers` and `MaxClients` to the desired number of processes.

Figure 9 shows the results of a comparison between the default dynamic mechanism and several static configurations. For the dynamic mechanism, we used a `MaxClients` setting of 1,024; for the static configurations, we evaluated values of 128, 256, 512, and 1,024. Figure 9 reveals that all of the tested static configurations obtained lower peak performance than the dynamic mechanism. This is because during the normal operating range, the dynamic configuration requires only a small number of processes, as shown in Figure 8(b). The static configurations all had higher numbers of processes, and thus had higher overhead to administer, resulting in lower peak performance. One additional observation from Figure 9 is that once an overload condition occurs, a larger number of server processes results in more predictable behaviour, as we noted earlier.

As a result of these observations, we believe that it is not a good idea to disable Apache's dynamic mechanism for controlling the number of server processes. Although there may be a static configuration that works well for a particular workload, we expect that the configuration will vary across different workloads. Thus is it likely simpler and as efficient to use the dynamic mechanism. We use the default dynamic mechanism for all of our remaining experiments.

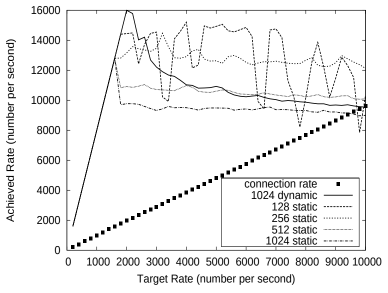


Figure 9. Effect of number of server processes on Apache performance

5.1.6. Persistent Connections and Pipelining

Two enhancements to the HTTP/1.1 protocol are persistent connections and pipelining [8]. These enhancements can potentially increase the performance of a Web server in several ways. First, a persistent TCP connection can be use to transfer multiple requests and responses, thus reducing the number of handshakes required to establish and terminate TCP connections. Persistent connections can also reduce the amount of TCP state that must be kept at the server, and improve throughput by increasing the volume of data that is transferred on each TCP connection. Pipelining can further improve performance. Pipelining allows multiple HTTP requests to be sent on a TCP connection without waiting for the corresponding HTTP responses to be received before sending the next request. This enhancement reduces the number of round trips that occur, which is the cause of the performance improvement.

In this section we explore the performance benefits of persistent connections and pipelining for the Apache Web server. Both of these features are enabled by default, so we did not have to alter the configuration of the server.

First, we examine the benefits of persistent connections alone. We consider reusing a TCP connection for either two, four, or eight requests for a 1 KB file. The results are shown in Figure 10(a). These results are for an Apache server using memory-mapping, and with 1,024 maximum processes (to smooth the performance under overload conditions). As we have already seen, the maximum performance of the server when using a separate TCP connection for each HTTP request is 7,300 responses per second. When each TCP connection transfers two requests and responses, the performance improves to 9,600 responses per second, an increase of 32%. The performance increases further as the amount of reuse increases. Using a TCP connection for four requests and responses increases the performance to 10,400 responses per second. 11,200 responses per second are supported when each TCP connection transfers eight requests and responses. Note however that the maximum number of TCP connections supported decreases as the amount of reuse increases. This occurs because server processes are assigned to a TCP connection for a longer period of time. As a result more processes are needed, which increases the overhead.

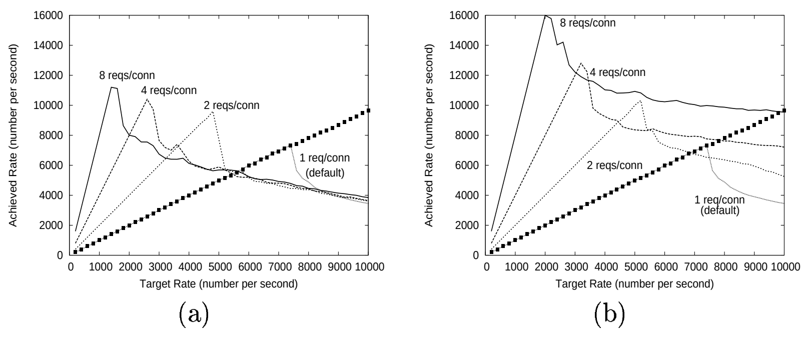


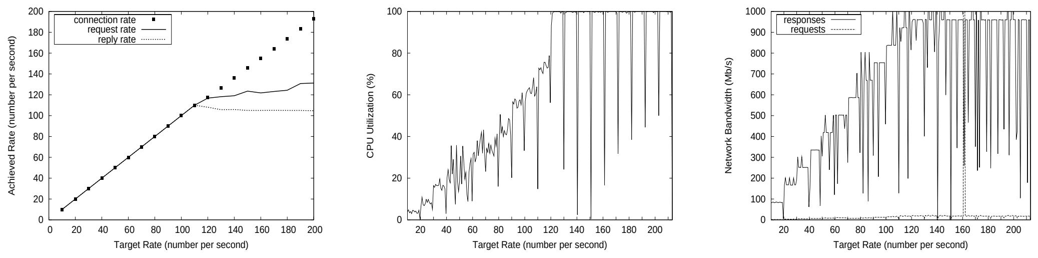
Figure 10. Effect of persistent connections and pipelining on Apache performance

Comparing Figure 10(b) to Figure 10(a) indicates the impact of using pipelining of requests in addition to persistent connections. As expected, utilizing pipelining increases the performance of the server. Using pipelining when issuing two requests on a connection results in a modest 4% improvement over persistent connections alone. Pipelining has more impact as the number of requests per connection increases. When eight requests for a 1 KB file are pipelined over each TCP connection, the server achieves a peak response rate of 16,000 per second. This is a 43% increase over using just persistent connections, and an increase of nearly 220% over non-persistent connections (i.e., when each HTTP request requires a separate connection). Pipelining provides significant gains over persistent connections alone, due to the asynchronous submission of requests.

Achieving similar gains in a WAN environment may be more challenging, primarily due to the larger latencies that will be experienced between the clients and the server. Server administrators may need to alter the configuration to clean up idle connections more aggressively when the server is busy. An evaluation of this type is beyond the scope of this paper. However, what should be obvious from the results presented here is that persistent connections and pipelining offer significant performance advantages, and thus should not be disabled.

5.1.7. Other Observations

Some of the factors that we examined during our Web server configuration study did not have a significant impact on server performance. For example, the `atime` system parameter, which controls the updating of a file's access time information, had little effect in our experiments, particularly when memory mapping of the file being served is used. Likewise, altering the length of the Apache server's listen queue (via `ListenBacklog`) had little impact, since it is not able to alleviate the CPU bottleneck. Both of these parameters could have an impact in other studies, if different workloads, file systems, and network environments are used.



(a) (b) (c)

Figure 11. Apache performance results for 1 MB transfers

There are many, many factors that can affect the performance of a Web server, far more than we could even attempt to analyze. In this paper, we focused our discussion on some of the more commonly discussed parameters for the Apache Web server.

5.2. Large transfers

In this section, we conduct experiments with 1 MB file transfers. Much lower request rates were used in these tests, with request rates between 20 and 200 per second.

Unlike the experiments with small transfers, the Apache server has no problem handling the request rates generated in these tests. We do not examine higher request rates, since these rates are sufficient for encountering a bottleneck elsewhere in our test environment, namely the network, as demonstrated in Section 4.4.

Figure 11(a) shows the connection, request and response rates achieved during the 1 MB tests. This figure reveals that once the network bottleneck occurs, the achieved request rate drops below the target rate. This occurs because there are not always processes available to accept the incoming requests, as processes are spending longer trying to respond to already accepted requests. The achieved response rates are even lower, as the network bottleneck dramatically slows the transfer rates. In an attempt to imitate actual user behaviour (i.e., user frustration resulting in aborted connections), we configure `httperf` to timeout a connection if a request or response takes too long. When a timeout occurs, the connection is reset. This further reduces the (successful) response rate.

Figure 11(b) indicates that the client CPU reaches 100% utilization once the network bottleneck has been encountered. Figure 11(c) shows that, as expected, the network link between the server and the Ethernet switch is completely utilized at a target response rate of 120 per second. As we discussed in Section 4.4, the CPU utilization reaches 100% after the network bottleneck occurs. Again, this points out the need for detailed data from multiple components in order to properly identify the system bottleneck.

Table III. Summary of Results

Factor	Impact	Section
access log	degrades performance slightly	5.1.2
`MaxRequestsPerChild`	can degrade performance significantly	5.1.3
`MMapFile`	improves performance	5.1.4
`MaxClients`	affects overload behaviour	5.1.5
persistent connections	improve performance	5.1.6
pipelining	significantly improves performance	5.1.6

5.3. Summary of results

This section presented our experimental results involving several workloads and numerous Web server configurations. Table III summarizes our results, and lists the sections which examined each of these factors.

Our recommendations regarding Web server configuration are:

- Logging of client requests incurs a modest penalty. If peak server performance is vital, then logging should be disabled.
- The creation and termination of server processes should be avoided when the server is busy. In particular, the `MaxRequestsPerChild` configuration option should not be changed from its default value of 0 unless it is absolutely necessary (e.g., to clean up known memory leaks).
- Memory mapping of files provides a reasonable performance improvement. However, there are disadvantages to be considered, such as identifying which files to map, and restarting the server if the files are modified.
- Apache is quite effective at creating extra processes when they are needed, and terminating them when they are not required. This (default) mechanism should only be disabled under exceptional circumstances.
- Persistent connections and pipelining offer significant increases in performance. These default features of the Apache server should not be disabled; rather, efforts should be focused on optimizing the use of these features.

In addition to these recommendations, our results also offered information on how to identify different bottlenecks. We discussed several situations where a bottleneck caused unexpected behaviours to occur elsewhere in the system. In these situations identifying the correct bottleneck may not be straightforward. This observation stresses the importance of monitoring many system components in a timely fashion.

6. CONCLUSIONS

In this paper, we evaluated the effects of different system and application configurations on the performance of a Web server. By monitoring many system components, we revealed how different bottlenecks can cause similar behaviours, and that some non-bottleneck resources behave in unexpected ways, which can make it difficult to identify the true bottleneck resource.

The work discussed in this paper was done as part of the validation process of our study of Web server benchmarking in an emulated WAN environment. The insights we gained on identifying bottlenecks helped us to verify various behaviours we observed in the emulated environment, where we did not have access to as much information. We believe that the approach used in this paper could help others to identify bottlenecks in other environments.

Our primary focus for future work is on identifying bottlenecks in more complex environments. For example, as part of our network emulation work, we need to examine more realistic workloads and to identify bottlenecks in WAN rather than LAN environments. A more ambitious goal is to (eventually) automate the identification of bottlenecks in benchmarking experiments.

ACKNOWLEDGMENTS

Financial support for this work was provided by iCORE (Informatics Circle of Research Excellence) in the province of Alberta, and by the Natural Sciences and Engineering Research Council (NSERC) of Canada. The authors are grateful to Nayden Markatchev and Rob Simmonds for their assistance with the test environment. The authors would also like to thank Tim Brecht, Balachander Krishnamurthy, and the anonymous reviewers for their constructive feedback.

REFERENCES

1. M. Abbott, "Making Apache Ten Times Faster". `http://da.teltecnz.co.nz/manual/apache-1.3.12/misc/perf-mja.html`
2. "The Apache HTTP Server Project". `http://httpd.apache.org/`
3. M. Aron and P. Druschel, "TCP Implementation Enhancements for Improving Web Server Performance", Technical Report TR99-335, Rice University, July 1999.
4. H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements", *Proceedings of IEEE INFOCOM*, pp. 252-262, San Francisco, CA, March 1998.
5. P. Barford and M. Crovella, "Measuring Web Performance in the Wide Area", *ACM Performance Evaluation Review*, Vol. 27, No. 2, pp. 35-46, September 1999.
6. P. Barford and D. Plonka, "Characteristics of Network Traffic Flow Anomalies", *Proceedings of the First ACM SIGCOMM Internet Measurement Workshop (IMW 2001)*, San Francisco, CA, pp. 69-73, November 2001.
7. N. Bansal and M. Harchol-Balter, "Analysis of SRPT Scheduling: Investigating Unfairness", *Proceedings of ACM SIGMETRICS Conference*, pp. 279-290, Cambridge, MA, June 2001.
8. R. Fielding, J. Gettys, J. Mogul, H. Frystyk-Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", RFC 2616, HTTP Working Group, June 1999.
9. D. Gaudet, "Apache Performance Notes". `http://httpd.apache.org/docs/misc/perf-tuning.html`
10. S. Godard, `systat` home page. `http://perso.wanadoo.fr/sebastien.godard/`
11. "`httperf` home page". `ftp://ftp.hpl.hp.com/pub/httperf`
12. J. Hu, S. Mungee, and D. Schmidt, "Techniques for Developing and Measuring High-Performance Web Servers over ATM Networks", *Proceedings of IEEE INFOCOM*, San Francisco, CA, March/April 1998.
13. Y. Hu, A. Nanda, and Q. Yang, "Measurement, Analysis, and Performance Improvement of the Apache Web Server", Technical Report No. 1097-0001, University of Rhode Island, 1997.

14. Information Networks Division, Hewlett-Packard Company, "netperf: A Network Performance Benchmark", Revision 2.0, `http://www.netperf.org/netperf/training/Netperf.html`
15. Information Networks Division, Hewlett-Packard Company, "The Public `netperf` Home Page". `http://www.netperf.org/`
16. R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling,* John Wiley & Sons, Inc., New York, NY, 1991.
17. D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance", *ACM Performance Evaluation Review*, Vol. 26, No. 3, pp. 31-37, December 1998.
18. E. Nahum, T. Barzilai, and D. Kandlur, "Performance Issues in WWW Servers", *IEEE/ACM Transactions on Networking*, Vol. 10, No. 1, pp. 2-11, February 2002.
19. E. Nahum, M. Rosu, S. Seshan, and J. Almeida, "The Effects of Wide-Area Conditions on WWW Server Performance", *Proceedings of ACM SIGMETRICS Conference*, Cambridge, MA, pp. 257-267, June 2001.
20. Netcraft Web Server Survey Archives. `http://news.netcraft.com/archives/web_server_survey.html`
21. M. Nielsen, "How to use a RAMdisk for Linux", `http://www.linuxfocus.org/English/November1999/article124.html`
22. V. Pai, P. Druschel, W. Zwaenepoel, "Flash: An efficient and portable Web server", *Proceedings of 1999 USENIX Annual Technical Conference*, Monterey, CA, pp. 199-212, June 1999.
23. Red Hat, "Red Hat Linux 8.0: The Official Red Hat Linux Reference Guide", `http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/ref-guide/`
24. A. Rousskov and D. Wessels, "High Performance Benchmarking with Web Polygraph", *Software Practice and Experience*, 2003.
25. C. Williamson, R. Simmonds, and R. Bradford, "A Case Study of Web Server Benchmarking Using Parallel WAN Emulation", *Proceedings of IFIP Performance 2002*, Rome, Italy, September 2002.
26. Standard Performance Evaluation Corporation, `www.spec.org`
27. Red Hat, "Tux Web Server Manuals", `www.redhat.com/docs/manuals/tux`