A Performance Comparison of Dynamic Web Technologies[1]

Lance Titchkosky[2], Martin Arlitt[3] and Carey Williamson

Abstract

Today, many Web sites dynamically generate responses "on the fly" when user requests are received. In this paper, we experimentally evaluate the impact of three different dynamic content technologies (Perl, PHP, and Java) on Web server performance. We quantify achievable performance first for static content serving, and then for dynamic content generation, considering cases both with and without database access. The results show that the overheads of dynamic content generation reduce the peak request rate supported by a Web server up to a factor of 8, depending on the workload characteristics and the technologies used. In general, our results show that Java server technologies typically outperform both Perl and PHP for dynamic content generation, though performance under overload conditions can be erratic for some implementations.

Keywords: Web Performance, Web Server Benchmarking, Dynamic Content Generation, Performance Evaluation

1 Introduction

On the World-Wide Web today, many sites dynamically create responses to user requests. Dynamic "on-the-fly" content creation provides Web site operators with several advantages: access to information stored in databases; the ability to personalize Web pages according to individual user preferences; and the opportunity to deliver a much more interactive user experience than possible with static Web pages alone.

Along with the advantages of dynamic content come several disadvantages. Dynamically generating Web content can significantly impact Web server performance, reducing the scalability of the Web site. Other disadvantages include security and availability concerns. Dynamically generated content can create security vulnerabilities or Denial-of-Service (DoS) opportunities, beyond those of static content Web sites.

In this paper, we examine the impacts on Web server performance from three different popular dynamic Web content technologies: Perl, PHP, and Java. The security and availability issues are beyond the scope of this paper.

Our experimental measurement results quantify the impacts of dynamic content generation on Web server performance. In particular, the overheads of database access and the processing required for dynamic content generation each take their toll. Combined, these effects reduce the peak request rate supported by a server up to a factor of 8, depending on the workload characteristics and the technologies used. In general, our results indicate that Java server technologies outperform both PHP and Perl, but there are many performance tradeoffs among these technologies. In particular, we find that Web server performance under overload can be quite erratic.

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

There have been numerous studies evaluating Web server performance. Many of these studies focus on Web server performance in LAN environments [5, 6, 9]. Several more recent studies consider Web server performance in WAN environments [2, 10, 18]. To the best of our knowledge, all of these studies only consider static Web content.

In 1995, Yeager and McGrath [19] studied the effects of dynamic content workloads on Web server performance. These results indicated that the Common Gateway Interface (CGI), which enables the dynamic generation of Web content, is much slower than directly retrieving a static file of the same size. Since this study, Web server architectures have improved significantly, and new technologies are now widely used for dynamic Web content generation. Thus our work can be compared to Yeager and McGrath's to determine how the performance of dynamic Web page generation has changed relative to static content serving in the past decade.

Cecchet *et al.* examine more recent dynamic Web content generation technologies, using two distinct benchmarks [3]. Our work is complementary to theirs. Based on the benchmarks they used, their results are of particular interest to online bookstore and auction sites. Our work, with simpler workloads, identifies more general performance tradeoffs relevant to any site using dynamic Web content generation.

There have also been many commercial Web server benchmarking studies. For example, many server companies use standard benchmarks such as SPECWeb to measure the performance of their products relative to those of their com-

[1] A 4 page version of this paper appears in [14].

[2] The authors are with the University of Calgary, Calgary, AB.

[3] Contact Author: arlitt@cpsc.ucalgary.ca

petitors. The current version of the SPECWeb benchmark, SPECWeb99[1], measures the performance of a Web server using requests for both static and dynamic content.

A second commercial benchmark of interest is TPC-W[2]. TPC-W is a transactional Web benchmark developed by the Transaction Processing Performance Council. The workload generated by TPC-W is intended to represent the activities of a business oriented transactional Web server. A detailed evaluation of TPC-W is provided in [4].

Developing realistic benchmarks requires detailed knowledge of the system workloads. One challenge for benchmark designers has been the scarcity of characterization studies for dynamic Web workloads. Studies to date include [1], [13], and [17]. As the understanding of dynamic Web workloads improves, so too will the realism of the benchmarks.

3 Experimental Environment

Our testbed consists of four IBM x335 servers: two for the clients, one for the Web server, and one for the database server. These machines are interconnected via a 1 Gbps full-duplex switched Ethernet LAN. The remainder of this section provides a detailed description of this test environment.

3.1 Hardware Configuration

3.1.1 Client Configuration: The two client machines in our testbed are rack-mounted IBM x335 servers running RedHat Linux 8.0. Each machine has a single Intel 2.4 GHz Intel Xeon processor, 1 GB of RAM, and a 36 GB 15K U320 SCSI disk. Each client used 124 MB of RAM as a "RAMdisk" (a virtual disk in memory [11]) for collecting statistics on the client's behaviour during testing. Each client machine has two 1 Gbps Ethernet NICs, although only one NIC on each machine is used in the experiments.

Several changes were made to the Linux kernel configuration on the clients. First, the number of available file descriptors was increased from 1,024 to 32,768. Second, we enabled TCP TIME_WAIT recycling. Both of these changes were necessary to allow the client to generate and sustain high request rates. Finally, all non-essential processes on the client machines were disabled, to minimize the consumption of resources by processes unrelated to workload generation.

3.1.2 Server Configuration: The server machines in our test environment are rack-mounted IBM x335 servers running RedHat Linux 7.3. The server hardware configuration is identical to that of the clients. The Web server also uses 124 MB of memory as a RAMdisk for storing statistics collected during tests. As with the client machines, all non-essential processes on the servers were disabled prior to conducting any tests, and available file descriptors were increased to 32,768.

3.1.3 Network Configuration: The client and server machines are connected to an HP Procurve 5300XL switch. This switch is configured with 20 full-duplex 1 Gbps ports.

3.2 Software Configuration

3.2.1 Client Workload Generation: For all of the tests in this study we use `httperf`, a tool for measuring HTTP performance [8]. 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. Although we only use a subset of `httperf`'s features in this work, we intend to use more of these capabilities in future work. Third, `httperf` supports open-loop workload generation, allowing the exploration of server performance under overload. Finally, `httperf` is available[3] in source code form, so that we can add additional functionality if desired.

3.2.2 Server Software: We use several different Web servers, modules, and servlet containers in our work:

- **Tux**

  Tux[4] is a kernel-based, multi-threaded, high-performance Web server available for Linux systems [15]. We use Tux version 2.1 to demonstrate that our client workload generators are not the bottleneck in any of our tests.

- **Apache**

  Many of our experiments involve the Apache[5] Web server. We use Apache server versions 1.3.27 and 2.0.45 in our work. Apache 1.3.27 is a process-based server that uses a separate process to handle each outstanding request. Apache 2.0.45 uses a hybrid thread and process model in an attempt to improve the server's performance. We include Apache in our tests because it is the most popular Web server on the Internet, used by more than 60% of all Web sites[6].

- **PHP**

  PHP (Hypertext Preprocessor)[7] is a scripting language specifically designed for use on the Web. It is the most popular dynamic Web content technology for use with Apache servers. According to an ongoing, automated survey, PHP is used by half of all Web sites running Apache [12]. PHP's popularity is due to its low cost (free) and its ease of use. Our tests use PHP version 4.3.1 compiled as a module for both Apache 1.3.27 and Apache 2.0.45.

[1] `http://www.spec.org/web99/`
[2] `http://www.tpc.org/tpcw/`
[3] `ftp://ftp.hpl.hp.com/pub/httperf`
[4] `http://people.redhat.com/mingo/TUX-patches/`
[5] `http://httpd.apache.org/`
[6] `http://news.netcraft.com/`
[7] `http://www.php.net/`

- **Perl**

  Perl[8] is a popular general purpose scripting language developed by Larry Wall in 1987. Perl was not designed to be a Web scripting language, but has been extended to include functionality useful for Web development. Perl is available under the GNU General Public License and an Artistic License, and thus is free to use. In early usage, the performance of Perl for dynamic content creation was quite slow, since a new Perl interpreter was spawned for each incoming Web request. To avoid this process creation overhead, an Apache module (`mod_perl`[9]) was created. This module embeds a persistent Perl interpreter into Apache itself. Approximately 20% of all Apache Web sites use `mod_perl` [12].

  Our Perl tests use Perl version 5.6.1 on Apache 1.3.27 with `mod_perl` version 1.27. We could not test Perl with Apache 2.0.45, since we were unable to install `mod_perl` 2.0 successfully on our server.

- **Server-Side Java**

  Server-side Java is a relatively new technology that uses a pool of Java virtual machines to respond to Web requests. It is a subset of Sun's Java 2 Enterprise Edition (J2EE)[10] technology. The servers, which run Java, are known as servlet containers or Java servers. We examine three servlet containers in this paper: Tomcat, Jetty, and Resin. Sun's Java Development Kit version 1.41.1_02 was used for all three of the tested servlet containers. We run the servlet containers in stand-alone mode, rather than integrating them into Apache.

  - **Tomcat**

    Tomcat[11] is a servlet container that provides the official reference implementation for both Java Servlets and Java Server Pages. For our work, we use Tomcat version 4.1.24.

  - **Jetty**

    Jetty[12] is a Web server and Java servlet container written entirely in Java. It is an open source project, but the majority of the development is done by Mort Bay Consulting. Jetty is advertised as one of the fastest servlet servers, which motivated us to include it in our testing. For our work, we use Jetty version 4.2.9.

  - **Resin**

    Resin[13] is a commercial Web server and Java servlet container that is freely available to individuals for non-commercial use. The Resin Web site claims that Resin's performance matches or exceeds that of Apache for static files. After evaluating several versions of the Resin server, we decided to use version 2.1.9 in our tests.

- **MySQL**

  MySQL is open source[14] database software, known for its high performance and reliability. We use MySQL version 4.0.12 in the experiments with database access.

We tuned each of the servers to provide as fair a comparison as possible. On all of the servers, the default per-request access log was disabled, though error logs were still used. In addition, the following changes were made, after evaluating the effects of different configuration parameters on the performance of each server. On Apache 1.3.27, the `MaxClients` parameter was set to 256 and the `MaxRequestsPerChild` was set to 0. On Apache 2.0.45, `MaxClients` was set to 250 (a multiple of the `ThreadsPerChild` parameter). With Tomcat, `enableLookups` (DNS) was disabled, and `maxProcessors` was set to 100. No additional changes were made to the configurations of the Tux, Jetty and Resin servers.

3.2.3 Monitoring Software: We use several sources of performance data to quantify the results of our experiments and to help identify bottlenecks. We use the `sar` (system activity report) utility[15] to monitor system resource utilization (e.g., CPU usage, I/O transactions, network utilization). `netstat` provides information on network-related errors such as the number of dropped TCP connections. The output of `httperf` includes numerous statistics on TCP and HTTP-level behaviour, including the average TCP connection rate, the HTTP request rate, and the HTTP reply rate. The Web server error logs indicate when problems occur with the server application (e.g., too many concurrent connections).

3.3 Controlling the Test Environment

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. Each experiment is specified as a shell script that is executed on the control machine. Controlling the experiments in this way ensures that the tests are conducted consistently. Archiving the scripts aids in repeating the results as well.

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

[8] `http://www.perl.com/`
[9] `http://perl.apache.org/`
[10] `http://java.sun.com/j2ee/`
[11] `http://jakarta.apache.org/tomcat/`
[12] `http://jetty.mortbay.org/jetty/`
[13] `http://www.caucho.com/resin/`
[14] `http://www.mysql.com/`
[15] `http://perso.wanadoo.fr/sebastien.godard/`

Table 1: Experimental Factors and Levels

Type	Factor	Levels
Client Workload Parameters	Response Size	2 KB, 64 KB
	Request Rate	200-6,000/second (2 KB); 200-2,000/second (64 KB)
	Response Type	static, dynamic, dynamic/database
Server	Software	Apache, Perl, PHP, Tomcat, Jetty, Resin

4 Performance Evaluation Methodology

We examine four factors in our experiments, using a one-factor-at-a-time experimental design [7]. Table 1 summarizes the factors and levels used in our experiments.

The first three factors listed in Table 1 describe the client workload. Since few characterization studies of dynamic Web workloads exist, we use a simple workload for our experiments. First, we issue requests for either a small file (2 KB) or a large file (64 KB). Second, we vary the request rate so that we can saturate the system and identify the bottleneck. The third factor is the response type, for which we examine three cases. Initially, we test the system using requests for static files. This "traditional" Web workload indicates the expected "best case" performance of the system. For the second level, the Web server dynamically generates a response of the requested size, using CPU resources but no I/O to the database. In the third case, the dynamic request results in a database access. Each HTTP request causes an SQL INSERT command that writes a small amount of data to the database. Then, 2 KB or 64 KB of text is generated, containing some data from an SQL SELECT command.

The final factor listed in Table 1 is the server software. These servers and modules were described in Section 3.2.2.

In this study we focus on evaluating server performance for several generic situations: requests for small or large static files, and requests requiring small or large dynamic responses, both with and without database access. We do not model individual user behaviour. Each client simply generates requests at the specified rate. Each TCP connection transfers at most one HTTP request and one HTTP response. A timeout value of 4 seconds is used to terminate TCP connections when the server is slow to respond. This enables us to generate and sustain overload conditions. Although the workload is simple, it still provides valuable insights into the performance of dynamic content generation. Using more realistic workloads to evaluate each of the servers remains as future work.

The run length for experiments was chosen subject to two constraints. First, the test duration must be long enough to assess accurately the server's ability to support the target request rate. Second, the duration of each test should be as short as possible, in order to test many request rates, and accurately identify the peak performance of each server.

We used a duration of 120 seconds for each test. As illustrated in Figure 1, this duration is sufficient for determining the server performance when the target response rate is below the server's peak rate. For example, the horizontal lines show that the tested server (Apache 2.0.45) exactly matches the target rates from 500 to 4,000 responses per second. Since the workload has only one unique file per test, the server's caching mechanism warms almost immediately, and performance is stable throughout the test. When more realistic workloads are used, such as in [3], longer durations are needed warm the server's caching mechanism and reach steady state.

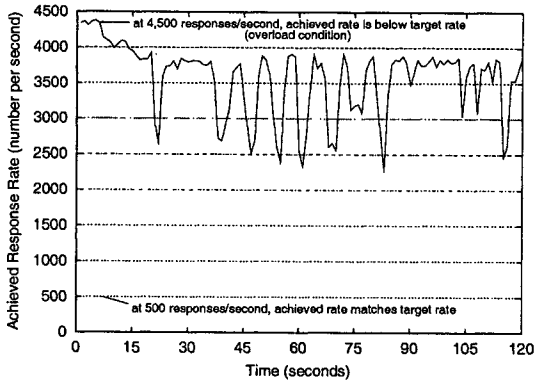


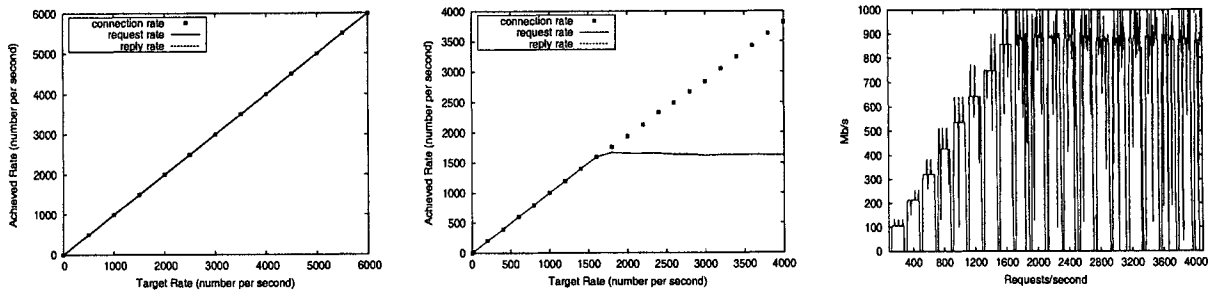
Figure 1: Apache 2.0.45 Response Rate Over Time (2 KB Static)

When the targeted response rate puts the server in an overload condition, it takes the server longer to reach "steady state". For example, Figure 1 shows that the server was not able to achieve the target rate of 4,500 responses per second. For the first few seconds of this test, the server comes close to achieving the target rate. However, shortly afterward the server performance degrades. After approximately 15 seconds, the server's performance fluctuates between 2,300 and 3,900 responses per second. While 120 seconds may not be long enough to determine the server's exact behaviour under overload, it is sufficient to know that the server is overloaded.

4.1 Validation of the Test Environment

In this section, we provide a basic "sanity check" of our experimental environment, in order to demonstrate its capacity. We conducted two experiments, one with a workload of 2 KB static files, the other with a workload of 64 KB static files. Both validation experiments use the Tux Web server.

Figure 2(a) shows the results for the experiment with 2 KB static files. This figure shows three sets of data. First, the points (black squares) represent the average number of TCP connections initiated by the clients during each 2-minute test.



(a) (b) (c)

Figure 2: Validation Tests Illustrating System Capacity: (a) 2 KB Static Files; (b) 64 KB Static Files; (c) Server Network Utilization

Second, the solid line (overlapping the points in this graph) shows the average rate at which HTTP requests were issued to the server. Third, the dashed line (also overlaid on the points in this graph) shows the average number of HTTP responses per second sent by the server in each test. The results provided in Section 5 focus on the average number of responses per second achieved by each tested server configuration.

The results in Figure 2(a) show that the two clients can generate and sustain a combined workload of 6,000 requests per second for a static 2 KB file. This figure also shows that with Tux, the server platform and the network are capable of supporting 6,000 responses per second for a static 2 KB file.

Figure 2(b) shows the results with 64 KB static responses for the Tux server. In this case, the system is limited to about 1,700 responses per second. While TCP connections are still established beyond that point, the server is unable to accept requests at a higher rate. The bottleneck in this case is the network between the server and the switch: on average, the server is transmitting approximately 900 Mbps of data, with peaks near 1 Gbps (1,000 Mbps), as shown in Figure 2(c). While we could have alleviated this bottleneck by using both network interfaces on the server, we decided not to do this since the achieved response rate already exceeds the expected range for any of the dynamic content servers. The results in Section 5 confirm this observation.

To summarize, our experimental infrastructure is capable of generating and sustaining request and response rates of (at least) 6,000 per second for 2 KB static files, and 1,700 per second for 64 KB static files. Achieved response rates lower than these in the main experiments indicate a bottleneck related to the particular server software being tested.

5 Experimental Results

5.1 Static Workloads

In this section, we examine the performance of the different Web servers for static Web content.

5.1.1 2 KB Static Workload: First, we examine the results for the 2 KB static files. Figure 3(a) shows the achieved response rates for the different Web servers in our experiments. The solid diagonal line represents the performance of the Tux server. As we discussed in Section 4.1, we use Tux to demonstrate the capacity of our testbed. Since the achieved response rate (y-axis) for Tux matches the target response rate (x-axis) for all tested rates, we know that our experimental infrastructure is capable of generating and sustaining (at least) 6,000 responses per second for 2 KB static files. Results lower than this indicate a bottleneck related to the particular server software being tested.

After Tux, Apache is the server with the best performance. Figure 3(a) shows that both Apache servers attain similar peak performance, supporting approximately 4,000 responses per second. Surprisingly, the Apache 2.0.45 server exhibits poorer performance than Apache 1.3.27 under overload (i.e., when the achieved rate is less than the target rate). That is, the performance drops off more sharply for Apache 2.0.45 than for Apache 1.3.27. Figure 3(b) shows the CPU utilization of each server by target response rate. Prior to overload, Apache 2.0.45 has a slightly higher CPU utilization than Apache 1.3.27 for equivalent response rates. Once the servers become overloaded, the CPU utilization of Apache 2.0.45 is lower than that of Apache 1.3.27. Under overload, Apache 2.0.45 is unable to accept TCP connections (and hence requests) as quickly as Apache 1.3.27. Figure 3(c) shows the number of failed TCP connection establishment attempts[16], a side effect from queues filling up when requests are not accepted quickly enough. Apache 2.0.45 has more failed TCP connections than Apache 1.3.27. Since Apache 2.0.45 accepts fewer connections when overloaded, it processes fewer requests than Apache 1.3.27, and has lower CPU utilization.

Figure 3(a) also shows the achieved response rates for the three Java-based Web servers, which all achieve significantly lower peak response rates than the Apache servers. Among the Java-based servers tested, Resin had the highest performance, peaking at 2,200 responses per second, followed by Tomcat (1,550 responses per second) and Jetty (1,150 responses per second). Since the Java-based servers are optimized for handling dynamic responses, they do not perform as well as the Apache servers for this static content workload.

[16] Attempts can exceed the target rate because of TCP retransmissions.

Figure 3(b) shows that the Jetty and Tomcat servers both reach 100% CPU utilization. The Resin server, however, reaches a peak CPU utilization of only 75%. The bottleneck in this case is the Java platform (version 1.4.1). In preliminary experiments with the next version (1.4.2), the Resin server achieves higher response rates, at a peak CPU utilization of 100%.

A final observation regarding the Java-based Web servers is that Tomcat and Jetty have much more predictable performance under overload than Resin. Figure 3(d) explains why. Figure 3(d) shows that as Jetty nears overload, it gradually increases the number of threads available for processing requests, until the maximum number of threads is reached. Jetty retains these threads for the rest of the experiment, providing relatively predictable performance under overload. Resin, on the other hand, behaves quite differently. It also spawns additional threads until the maximum number of threads is reached. However, unlike Jetty, Resin periodically terminates old threads and creates new ones. This overhead affects the performance of the Resin server under overload, making it less predictable than Jetty. Tomcat (not shown) behaves similarly to Jetty under overload, though Tomcat spawns new threads occasionally, which impacts performance somewhat.

5.1.2 64 KB Static Workload: Figure 4(a) shows the achieved response rates for the 64 KB static workload. As was discussed in Section 4.1, Tux achieves a maximum response rate near 1,700 responses per second. Beyond this rate, the network is the bottleneck. Apache 1.3.27 achieves a peak rate of 1,400 responses per second; the achieved rate then decreases slowly under overload. Apache 2.0.45 peaks near 1,300 responses per second. With this workload, Apache 2.0.45 behaves better under overload than it did for smaller files. The three Java-based servers again have much lower peak performance than the Apache servers. Jetty peaks near 700 responses per second, and degrades gracefully under overload. Resin supports a similar response rate, but performs more erratically under overload. For this workload, Tomcat has the poorest performance of all the servers evaluated, supporting only 450 responses per second. Tomcat's performance under overload is also somewhat erratic.

Figure 4(b) shows the CPU utilization of each server for the 64 KB static workload. All servers have a peak CPU utilization of 100%. The CPU utilizations during overload vary, for the same reasons that were discussed in Section 5.1.1.

5.2 Dynamic Workloads (without database access)

In this section, we analyze the performance of different dynamic content generation technologies, for small and large dynamic pages. In both cases, the content generation process does *not* involve database access.

5.2.1 2 KB Dynamic Workload (without database access): Figure 5(a) shows the achieved response rates for six different methods of dynamic content generation, for 2 KB responses. First, we examine the results that involve the Apache Web servers. The results in Figure 5(a) show that the peak performance of PHP on Apache 1.3.27 is 1,400 responses per second. This rate is about 35% of the performance of the Apache server for static content of the same size. PHP on Apache 2.0.45 has even poorer performance, peaking at 1,000 responses per second, or 25% of its peak performance for static 2 KB workloads. In other words, dynamic page generation alone can reduce the server's peak response rate by a factor of 3 to 4. Both PHP server configurations are stable under overload. The performance of Perl is similar to that of PHP on Apache 2.0.45. Figure 5(b) reveals that the PHP and Perl servers on Apache are all CPU-bound.

When generating content dynamically, the three Java-based servers all perform reasonably well compared to the PHP and Perl configured servers. Peak response rates of 1,300, 2,200 and 1,500 per second are achieved by Jetty, Resin, and Tomcat, respectively. These are comparable to PHP on Apache 1.3.27, and significantly better than PHP on Apache 2.0.45 or Perl on Apache 1.3.27. Surprisingly, the performance of the Java-based servers when generating 2 KB dynamic responses is almost identical to their performance when serving static files. This supports our observation that the Java-based servers are optimized for handling dynamic Web workloads. However, the Java-based servers behave more erratically under overload conditions than the PHP and Perl server configurations, for this particular workload. Figure 5(b) indicates that Jetty and Tomcat are CPU-bound, while Resin is not. Resin's performance is impeded by the poor scalability of the Java 1.4.1 platform [16], as indicated in Section 5.1.1.

Cecchet *et al.* observed that Java servlets were less efficient than PHP, due to inter-process communication overheads [3]. In our experiments, the Java servers operate in standalone mode, avoiding this overhead.

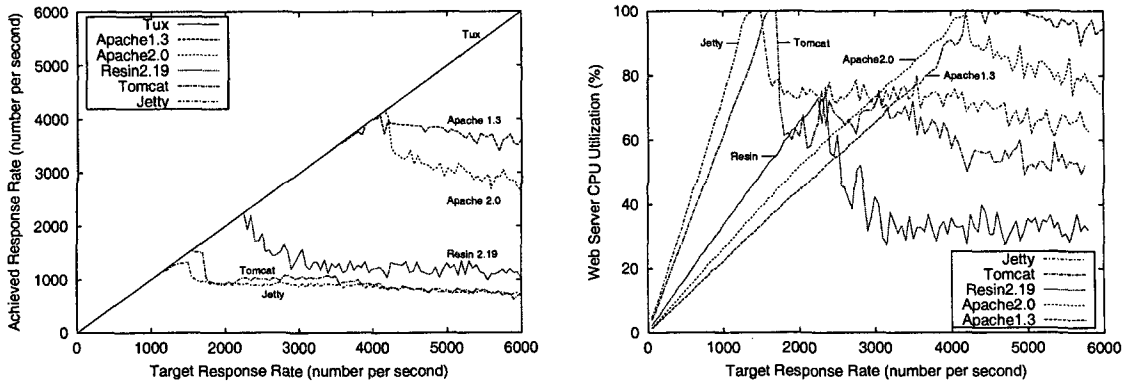
5.2.2 64 KB Dynamic Workload (without database access): The results of the experiments with 64 KB dynamic responses (without database access) are shown in Figure 6(a). Once again, the PHP-enabled servers have the lowest performance, supporting only 250 and 200 responses per second, respectively. These rates are less than 20% of the Apache server performance results for static files of the same size. Perl does significantly better than PHP, achieving a peak rate of 650 responses per second.

The Jetty and Tomcat servers achieve 450 and 400 responses per second, respectively, which is significantly lower than their performance when serving 64 KB static files. The Resin server achieves slightly higher performance, reaching a peak of 600 responses per second.

Figure 6(b) shows the CPU utilization results. The servers are all CPU-bound, as was the case for 64 KB static responses.

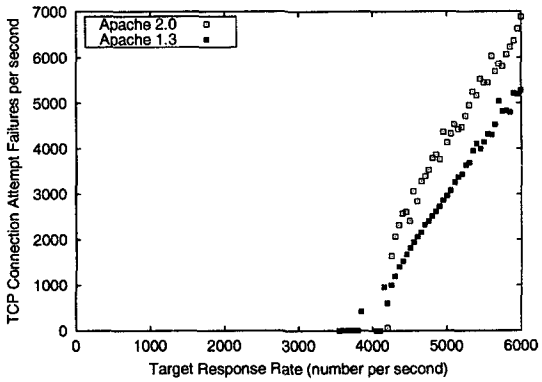
5.3 Dynamic Workloads (with database access)

In this section, we analyze the performance of the different dynamic content generation strategies for small and large dy-

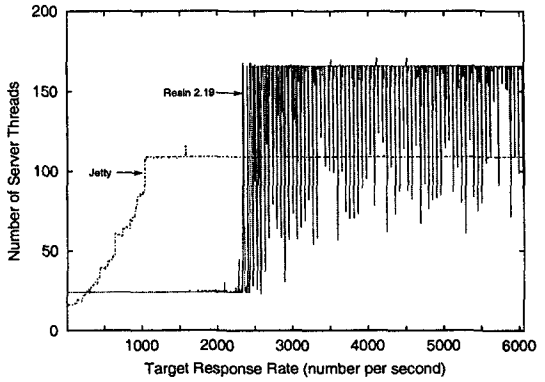


(a) Response Rates

(b) CPU Utilization

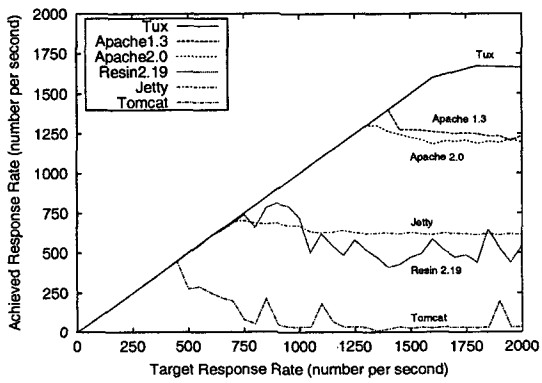


(c) Failed TCP Connection Attempts

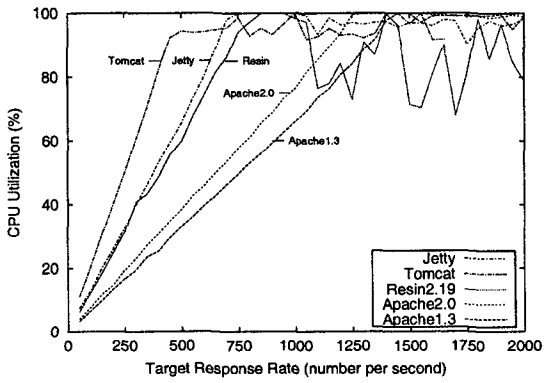


(d) Active Server Threads

Figure 3: Results for 2 KB Static Responses

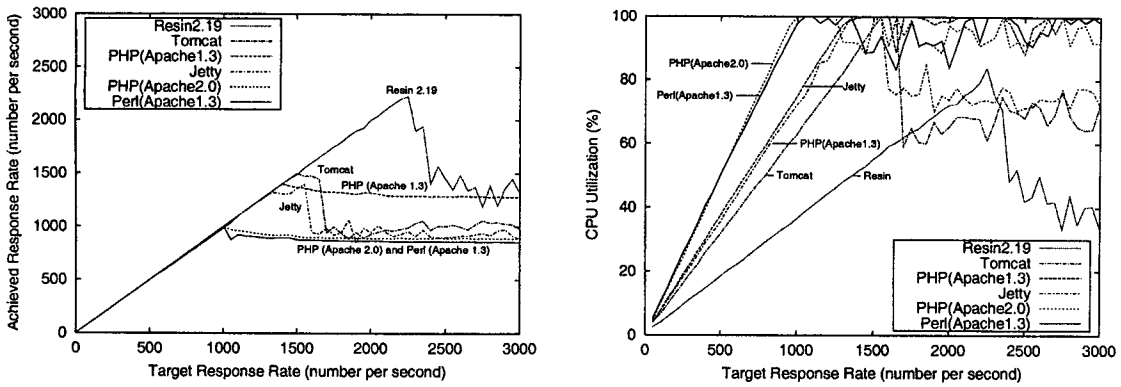


(a) Response Rates



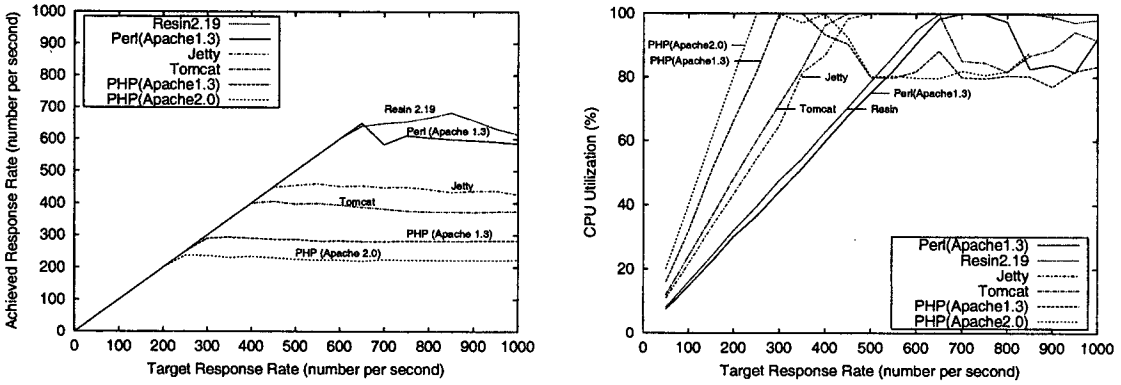
(b) CPU Utilization

Figure 4: Results for 64 KB Static Responses



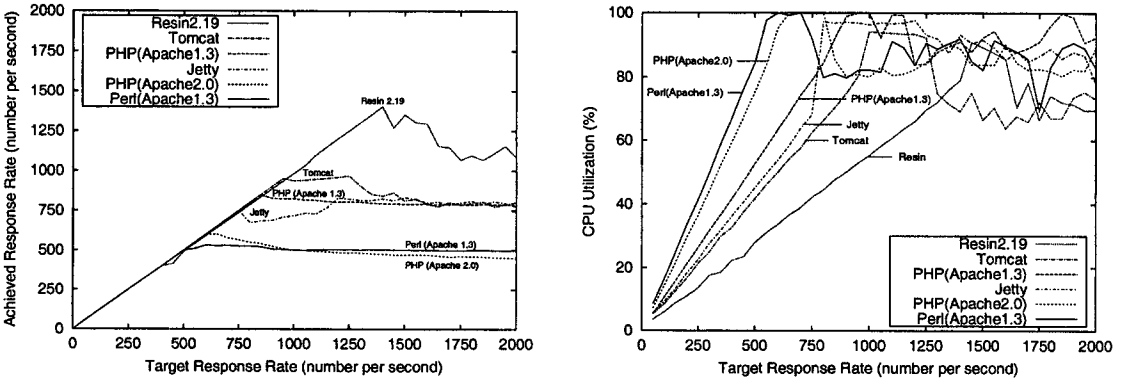
(a) Response Rates (b) CPU Utilization

Figure 5: Results for 2 KB Dynamic Responses (No DB Access)



(a) Response Rates (b) CPU Utilization

Figure 6: Results for 64 KB Dynamic Responses (No DB Access)



(a) Response Rates (b) CPU Utilization

Figure 7: Results for 2 KB Dynamic Responses (with DB Access)

namic pages. In these experiments, a single SQL INSERT and a single SQL SELECT command are executed when generating the response. We utilize a single persistent connection for all communication between the Web servers and the database server. This reduces the overhead of communication with the database, and avoids locking contention at the database.

5.3.1 2 KB Dynamic Workload (with database access): Figure 7(a) shows the results for the 2 KB dynamic responses requiring database access. Two observations are evident from this graph. First, the three Java-based servers outperform the servers that are using PHP or Perl. Second, accessing the database significantly reduces the performance of all servers. The peak performance of the servers in these experiments is 50% to 64% of that when no database access is required for dynamic content generation. In other words, database access can reduce the server response rate by another factor of 2.

Among the three Java-based servers, Resin has the highest peak performance at 1,400 responses per second, followed by Tomcat at 950 responses per second, and Jetty at 750 responses per second. All of these servers perform erratically under overload.

For the Apache-based servers, PHP on Apache 1.3.27 outperforms the others, with a peak performance of 850 responses per second. Next is PHP on Apache 2.0.45, which achieves 600 responses per second. Perl on Apache 1.3.27 has the lowest peak performance for this workload, supporting only 500 responses per second. This is 8 times fewer responses per second than is supported by Apache 1.3.27 for the 2 KB static workload. All of the Apache-based servers have relatively stable performance under overload.

Figure 7(b) suggests that the bottleneck in these experiments is not the Web server CPU. The actual bottleneck is the Web server threads/processes synchronizing over the single persistent connection to the database. This bottleneck could be alleviated with multiple persistent connections. However, a tradeoff exists between synchronizing requests on one or more persistent connections and lock contention at the database. Identifying the optimal number of persistent connections to use with each Web server is beyond the scope of this paper.

5.3.2 64 KB Dynamic Workload (with database access): The results of the experiments for 64 KB dynamic responses requiring database access are shown in Figure 8. As was the case with 2 KB responses, the Java-based servers performed at least as well as, and usually better than, the servers configured with PHP. Jetty and Tomcat peak around 350 responses per second, and behave well under overload. Resin again achieves the highest peak performance (approximately 550 responses per second), but is slightly more erratic under overload. Perl on Apache 1.3.27 achieves the second highest response rate of all the tested server configurations, supporting 400 responses per second. The PHP servers on Apache 1.3.27 and 2.0.45 had the worst peak performance, at 250 and 150 responses per second, respectively. Figure 8(b) shows that all of the Web servers are CPU-bound, as was the case for the other 64 KB experiments.

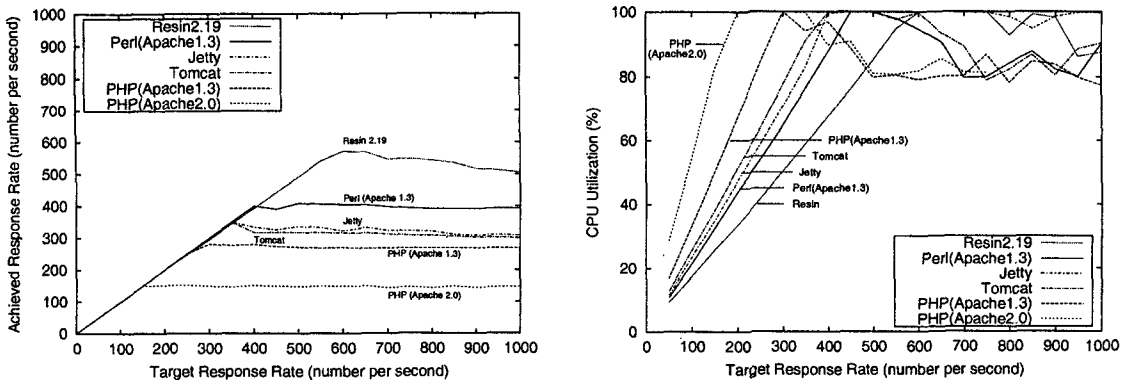
6 Summary and Conclusions

This paper presents a benchmarking study of dynamic content generation techniques. The experimental study is conducted using clients and servers in a dedicated Gigabit Ethernet LAN environment. To the best of our knowledge, this is the first study to evaluate such a broad range of dynamic content technologies using a variety of Web server software. While our study is far from comprehensive, we believe that it provides a state-of-the-art look at the performance tradeoffs between different technologies for dynamic Web content generation.

There are three main conclusions from this work. First, the ongoing trend toward personalization of Web content comes at a price. There is often a dual impact on Web server performance, from the overhead for database access, and from the processing required for dynamic content generation itself. Our experiments have quantified each of these effects. Combined, these effects reduce up to a factor of 8 the peak request rate supported by a server. Second, today's technologies for dynamic Web content generation exhibit distinct tradeoffs in terms of Web server performance. PHP handles small dynamic content requests well, but struggles with large dynamic content requests. Jetty, Resin, and Tomcat are ill-suited for serving static content, but perform better for their intended purpose of dynamic content generation. In general, our results indicate that Java server technologies outperform both PHP and Perl. Finally, Web server performance under overload can be quite unpredictable. Some dynamic content generation technologies are quite robust under overload (PHP, Perl, Jetty), while some are not (Resin). Consideration of overload behaviour may be just as important as the peak request rate when Web site administrators are choosing dynamic Web content generation technologies.

The results in this paper provide Internet practitioners and Web content developers with an indication of the performance tradeoffs associated with current technologies for dynamic content generation. Our benchmarking methodology is also useful for the evaluation of emerging Web technologies, and the performance tuning of existing implementations.

Our future work is focusing on characterizing dynamic content usage in academic and commercial Web sites, and on benchmarking Web server performance for more realistic dynamic content workloads. The performance impact of operating system and Java platform enhancements, as well as the synchronization and lock contention issues, also require further examination.



(a) Response Rates (b) CPU Utilization

Figure 8: Results for 64 KB Dynamic Responses (with DB Access)

Acknowledgements

Financial support for this research was provided by iCORE (Informatics Circle of Research Excellence), NSERC (Natural Sciences and Engineering Research Council), and CFI (Canada Foundation for Innovation). The authors are grateful to the anonymous reviewers, and to Nayden Markatchev for setting up the testbed in the ELISA lab.

References

[1] M. Arlitt, D. Krishnamurthy, and J. Rolia, "Characterizing the Scalability of a Large Web-based Shopping System", *ACM Transactions on Internet Technology*, Vol. 1, No. 1, pp. 44-69, August 2001.

[2] 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.

[3] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content", *Proceedings of* 4^{th} *Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[4] D. García and J. García, "TPC-W E-Commerce Benchmark Evaluation", *IEEE Computer*, Vol. 36, No. 2, pp. 42-48, February 2003.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] E. Nahum, M. Rosu, S. Seshan, and J. Almeida, "The Effects of Wide-Area Conditions on WWW Server Performance", *Proceedings of ACM SIGMETRICS*, Cambridge, MA, pp. 257-267, June 2001.

[11] M. Nielsen, "How to use a RAMdisk for Linux", `http://www.linuxfocus.org/English/November1999/article124.html`

[12] SecuritySpace, "Apache Module Report", April 2003, `http://www.securityspace.com/s_survey/data/man.200308/apachemods.html`.

[13] W. Shi, R. Wright, E. Collins, and V. Karamcheti, "Workload Characterization of a Personalized Web Site – And Its Implications for Dynamic Content Caching", *Proceedings of the* 7^{th} *International Workshop on Web Caching and Content Distribution (WCW '02)*, Boulder, CO, August 2002.

[14] L. Titchkosky, M. Arlitt, and C. Williamson, "Performance Benchmarking of Dynamic Web Technologies", *Proceedings of IEEE MASCOTS 2003*, October, 2003.

[15] Red Hat, "Tux Web Server Manuals", `http://www.redhat.com/docs/manuals/tux`

[16] Sun Microsystems, "Java 2 Platform, Standard Edition (J2SE), Version 1.4.2 Performance White Paper", `java.sun.com/j2se/1.4.2/1.4.2_whitepaper.html`

[17] U. Vallamsetty, K. Kant, and P. Mohapatra, "Characterization of E-Commerce Traffic", *Electronic Commerce Research Journal*, Vol. 3, No. 1-2, January/April 2003.

[18] C. Williamson, R. Simmonds, and M. Arlitt, "A Case Study of Web Server Benchmarking Using Parallel WAN Emulation", *Performance Evaluation*, Vol. 49, No. 1-4, pp. 111-127, September 2002.

[19] N. Yeager and R. McGrath, *Web Server Technology: The Advanced Guide for World Wide Web Information Providers,* Morgan-Kaufmann Publishers, Inc., San Francisco, CA, 1996.