Performance Benchmarking of Dynamic Web Technologies

Lance Titchkosky   Martin Arlitt   Carey Williamson
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB, Canada T2N 1N4
Email: {lancet,arlitt,carey}@cpsc.ucalgary.ca

May 8, 2003

Abstract

When the World-Wide Web was first created, the content on most Web sites was simply a collection of static files. Today, many Web sites dynamically generate responses "on the fly" when user requests are received. Dynamic creation of content provides a Web interface to information stored in databases, enabling the personalization of pages according to individual user preferences, and delivering a much more interactive Web browsing experience than is possible with static Web pages.

One disadvantage of dynamically generating Web content is the impact on Web server performance. In this paper, we experimentally evaluate the impact of three different dynamic content technologies 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 typically reduce by one half the peak request rate supported by a Web server. 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 Servers, Web Benchmarking, Web Performance, Dynamic Content Generation, Performance Evaluation Methodology

1 Introduction

On the World-Wide Web today, many sites dynamically create responses to user requests. Dynamic "on-the-fly" creation of content provides Web site operators with numerous advantages over content created entirely from static files. These advantages include 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 come several disadvantages. Dynamically generating Web content can significantly impact Web server performance. This can dramatically reduce 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 associated with static content Web sites.

In this paper, we examine the impact of three different dynamic Web content technologies on Web server performance. The security and availability issues are beyond the scope of this paper and are not discussed further. We examine three of the most popular dynamic Web technologies: Perl, PHP, and Java.

Our results serve to 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 result in the halving (or worse) of the peak request rate supported by a server. 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 [3, 4, 8]. Several more recent studies consider Web server performance in WAN environments [1, 9, 13, 16]. To the best of our knowledge, all of these studies only consider static Web content.

In 1995, Yeager and McGrath [17] studied the effects of dynamic content workloads on Web server performance. His 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 his 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 McGrath's to determine how the performance of dynamic Web page generation has changed relative to static content serving in the past decade.

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 competitors. The current version of the SPECWeb benchmark, SPECWeb99 [14], measures the performance of a Web server using requests for both static and dynamic content as part of the generated workload. The addition of dynamic content into SPECWeb99 is an obvious indication of the growing use of dynamic content on Web sites, and of the importance to understand the performance of Web servers serving dynamic content.

3 Experimental Environment

Our testbed consists of two clients submitting requests to a single Web server over a 1 Gbps full-duplex switched Ethernet LAN. The remainder of this section provides a detailed description of this test environment. Section 3.1 presents the hardware configuration of our test environment, while Section 3.2 introduces the software we used.

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 machine in our test environment is a rack-mounted IBM x335 server running RedHat Linux 7.3. The server hardware configuration is identical to that of the clients. This 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 server were disabled prior to conducting any tests. We also increased the number of available file descriptors to 32,768 and enabled TCP TIME_WAIT recycling.

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. No other machines were on this LAN during the experiments.

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 [7]. 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` is available[1] 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. In this section we introduce each package and describe how we use it.

- **Tux**

  Tux[2] is a kernel-based, multi-threaded, high-performance Web server available for Linux systems [15]. We use Tux version 2.1 to verify that our client workload generators are not the bottleneck in any of our tests.

- **Apache**

  Many of our experiments involve the Apache[3] 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 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 [10].

- **PHP**

  PHP (Hypertext Preprocessing)[4] 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 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] `ftp://ftp.hpl.hp.com/pub/httperf`

[2] `http://people.redhat.com/mingo/TUX-patches/`

[3] `http://httpd.apache.org/`

[4] `http://www.php.net/`

- **Perl**

  Perl[5] 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`[6]) was created. This module embeds a persistent Perl interpreter into Apache itself. `mod_perl` is used by approximately 20% of all Web sites that run Apache servers [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)[7] 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[8] 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[9] 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 [6], which is why we chose to include it in our testing. For our work, we use Jetty version 4.2.9.

  - **Resin**

    Resin[10] 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 [2]. We use Resin version 3.0.0 beta in our tests.

[5] `http://www.perl.com/`
[6] `http://perl.apache.org/`
[7] `http://java.sun.com/j2ee/`
[8] `http://jakarta.apache.org/tomcat/`
[9] `http://jetty.mortbay.org/jetty/`
[10] `http://www.caucho.com/`

- **MySQL**

  MySQL is open source[11] database software, known for its high performance and reliability. We use MySQL version 4.0.12 in the experiments with database access. The database software is run on the same platform as the Web server.

We did attempt to tune each of the servers in order to provide as fair a comparison as possible. On all of the servers, (per-request) logging was disabled (error logs were left on). 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[12] 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

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. 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 mechanism communicates (via `ssh`) with each machine involved in the experiment. Before starting the initial test, information is collected 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.

[11] `http://www.mysql.com/`

[12] `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	200-5,000/second (2 KB); 200-2,000/second (64 KB)
	Response Type	static, dynamic, dynamic/database
Server	Software	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 [5]. 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 what the bottleneck is. 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 "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 outputted, which contains data from an SQL SELECT command.

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

We run each of our tests for 120 seconds. We found that this was sufficiently long to assess system stability, yet short enough to permit the large number of tests needed for our study.

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. In both experiments we used Tux as the Web server.

Figure 1(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 two minute test. Second, the solid line (which overlaps the points in this graph) shows the average rate at which HTTP requests were issued to the server. Third, the dotted line (also overlaid on the points in this graph) shows the average number of HTTP responses per second sent by the server in each two minute test. Graphs of this form are used throughout the paper to illustrate the performance results.

The results in Figure 1(a) indicate that the two clients are able to generate and sustain a combined workload of 5,000 requests per second for a static 2 KB file. The achieved request rate

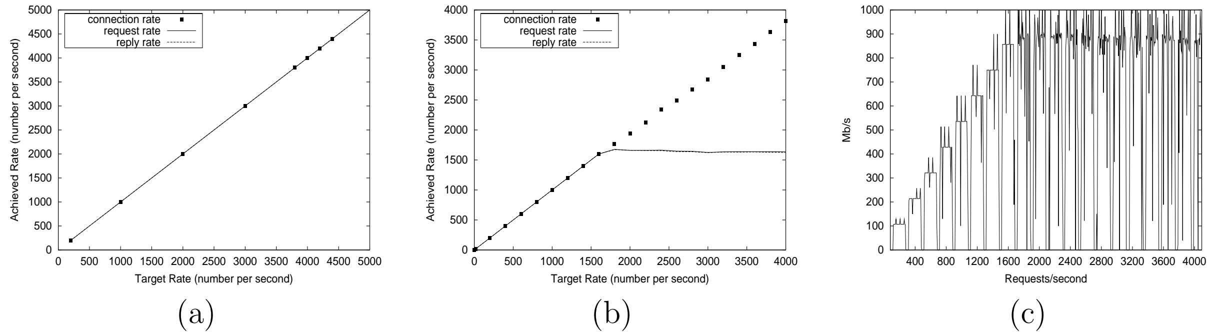


Figure 1: Validation Tests Illustrating System Capacity: (a) 2 KB Static Files; (b) 64 KB Static Files; (c) Server Network Utilization

matches the target request rate for all tests. This figure also indicates that with Tux, the server platform and the network are also capable of supporting 5,000 requests per second for a static 2 KB file.

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

To summarize, our experimental infrastructure is capable of generating and sustaining request rates of (at least) 5,000 requests per second for 2 KB static files, and 1,700 requests per second for 64 KB static files. Achieved request rates lower than these in the main experiments indicate a bottleneck related to the particular server software being used.

5 Experimental Results

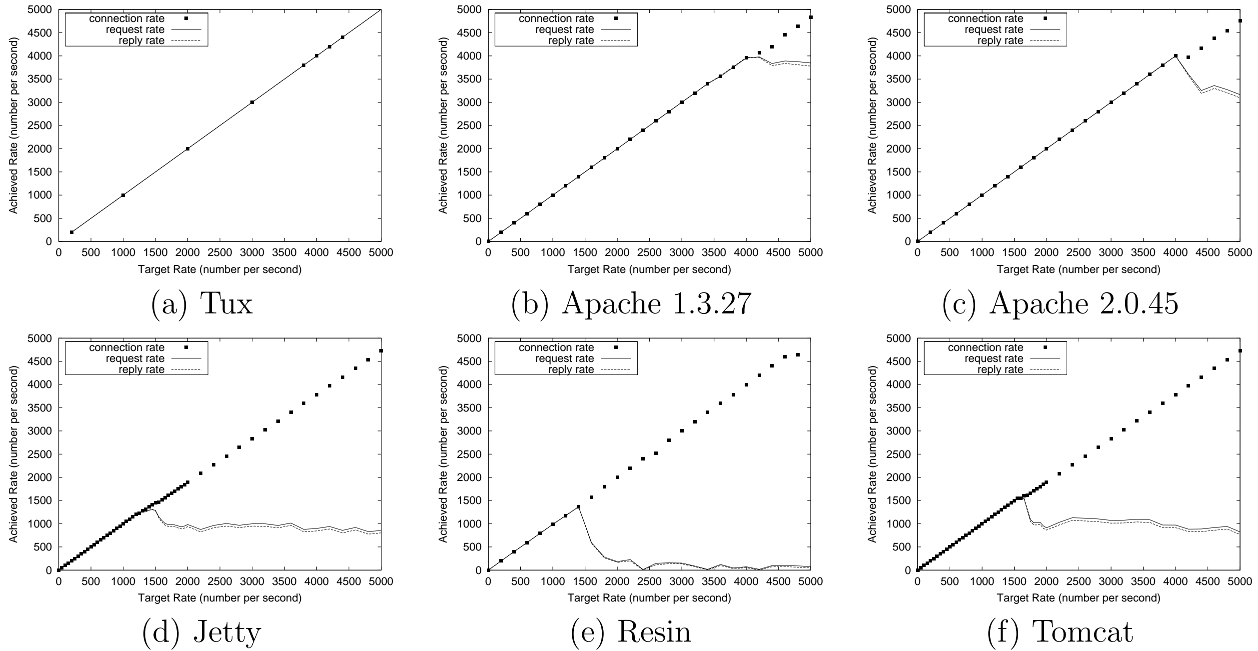
In this section we present the results of our experiments.

5.1 Static Workloads

In this section, we examine the performance of the different Web servers for static Web content. Section 5.1.1 presents the results for the 2 KB static files. Section 5.1.2 provides the results for the 64 KB static files.

5.1.1 2 KB Static Workload

Figure 2 shows the results for the different Web servers in our experiments. In order to simplify comparisons, all of the graphs in the figure use the same scales for the X and Y axes.



(a) Tux (b) Apache 1.3.27 (c) Apache 2.0.45

(d) Jetty (e) Resin (f) Tomcat

Figure 2: Experimental Results for 2 KB Static Responses

Figure 2(a) shows the results for the Tux Web server. As previously discussed, the Tux server demonstrates the capability of our test environment. Figure 2(a) indicates that over 5,000 requests per second are possible for a static 2 KB file.

Figures 2(b) and (c) show the results for the Apache 1.3.27 and 2.0.45 servers, respectively. Both servers perform similarly for this workload, supporting approximately 4,000 requests per second. Suprisingly, the Apache 2.0.45 server appears to have worse behaviour under overload (i.e., at request rates higher than the maximum supported rate). Its performance drops off more quickly than that for the Apache 1.3.27 server.

Figures 2(d)-(f) show the results for the three Java-based Web servers. All three of these servers have significantly lower peak request rates than the Apache servers. Among the Java-based servers tested, Tomcat had the highest performance, peaking at 1,550 requests per second, followed by Resin (1,400 requests per second) and Jetty (1,200 requests per second).

5.1.2 64 KB Static Workload

Figure 3 shows the results for the 64 KB static workload. As mentioned in Section 4.1, Tux supports a maximum request rate of approximately 1,700 requests per second (Figure 3). Beyond this limit, the network is the bottleneck.

Figures 3(b) and (c) show the performance results for the Apache servers. Apache 1.3.27 achieves a peak rate of 1,400 requests per second, then decreases slowly under overload. Apache 2.0.45 peaks near 1,300 requests 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

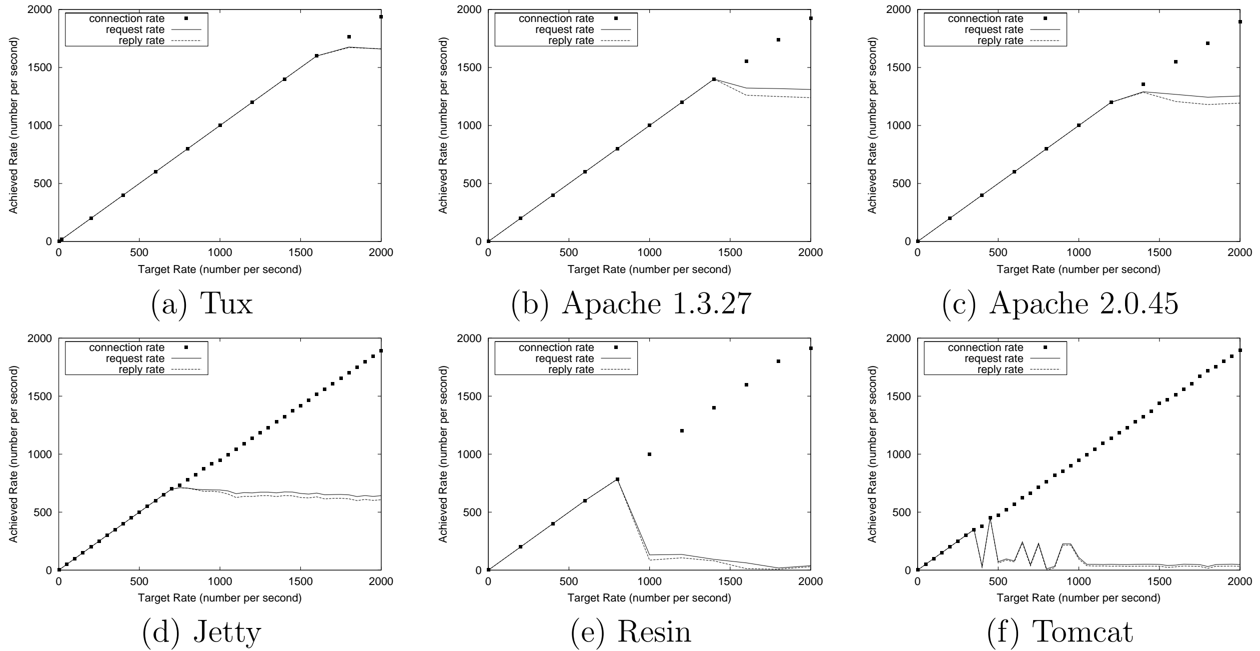


Figure 3: Experimental Results for 64 KB Static Responses

servers. Jetty peaks near 700 requests per second, and degrades gracefully under overload. Resin supports a higher request rate (800 per second) than Jetty, but performs quite poorly under overload. For this workload, Tomcat has the poorest performance of all the servers evaluated, supporting only 350 requests per second. Tomcat's performance under overload is quite erratic with this workload.

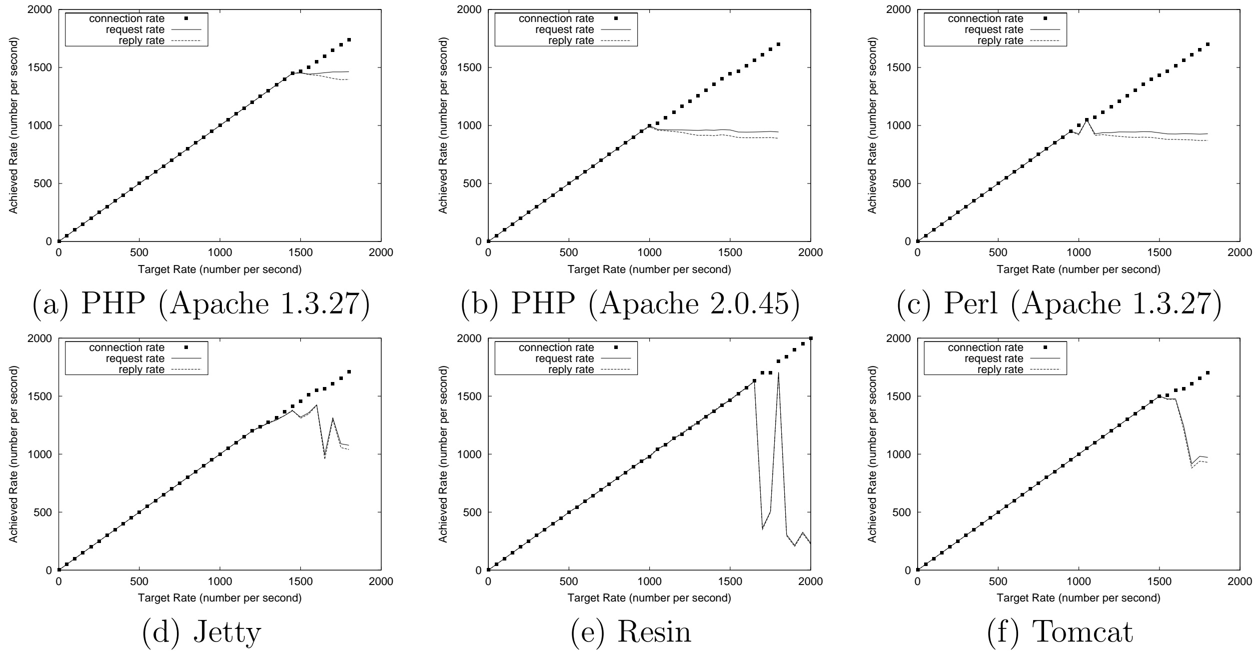
5.2 Dynamic Workloads (without database access)

In this section, we analyze the performance of different dynamic content generation technologies, under two different workloads. In both cases, the content generation process does *not* involve database access. Section 5.2.1 provides the results for 2 KB dynamic responses, while Section 5.2.2 presents the results for 64 KB dynamic responses.

5.2.1 2 KB Dynamic Workload (without database access)

Figure 4 shows the results for six different methods of dynamic content generation. Figure 4(a) presents the results for PHP running on Apache 1.3.27. The peak performance for this combination is 1,450 requests per second. This rate is about 37% of the performance of the Apache server for static content similar in size. PHP on Apache 2.0.45 (Figure 4(b)) has even poorer performance, peaking at 950 requests per second. Both PHP server configurations are stable under overload. Figure 4 shows that the performance of Perl mimics that of PHP on Apache 2.0.45.

When generating content dynamically, the three Java-based servers all perform reasonably



(a) PHP (Apache 1.3.27) (b) PHP (Apache 2.0.45) (c) Perl (Apache 1.3.27)

(d) Jetty (e) Resin (f) Tomcat

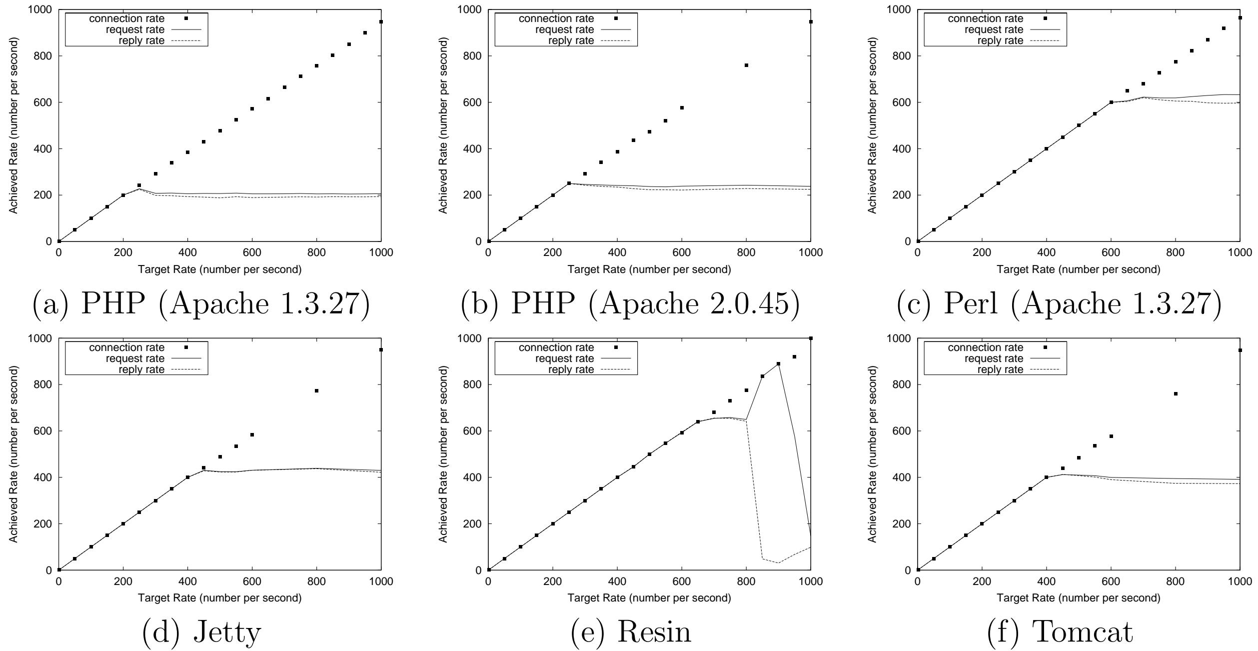
Figure 4: Experimental Results for 2 KB Dynamic Responses (No Database Access)

well compared to the PHP and Perl configured servers. Peak response rates of 1,200, 1,650 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. Even more surprising is the fact that Resin performed slightly better in this case than when serving static files. One final observation is that the Java-based servers do not appear to work as well under overload conditions as the PHP and Perl server configurations, for this particular workload.

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 5. The PHP-enabled servers have the lowest performance, supporting only 200 and 250 requests per second, respectively. These rates are about 20% of the Apache server performance results for static files of the same size. Perl does significantly better than PHP, achieving a peak of 600 requests per second.

The Jetty and Tomcat servers both achieve 400 requests per second, which is significantly lower than their performance when serving 64 KB static files. The Resin server again achieves slightly better performance for dynamic content than for static content, reaching a peak of 650 requests per second. However, the Resin server's behaviour under overload is quite different from any of the other servers. At a target rate of 800 requests per second, the Resin server is sluggish in responding to requests, while apparently shifting emphasis to accepting incoming connections. The large divergence between the request rate and the reply rate is seen only for this server.



(a) PHP (Apache 1.3.27) (b) PHP (Apache 2.0.45) (c) Perl (Apache 1.3.27)

(d) Jetty (e) Resin (f) Tomcat

Figure 5: Experimental Results fro 64 KB Dynamic Responses (No Database Access)

Resin is able to accept request rates as high as 900 per second. At request rates beyond this, Resin's performance degrades sharply.

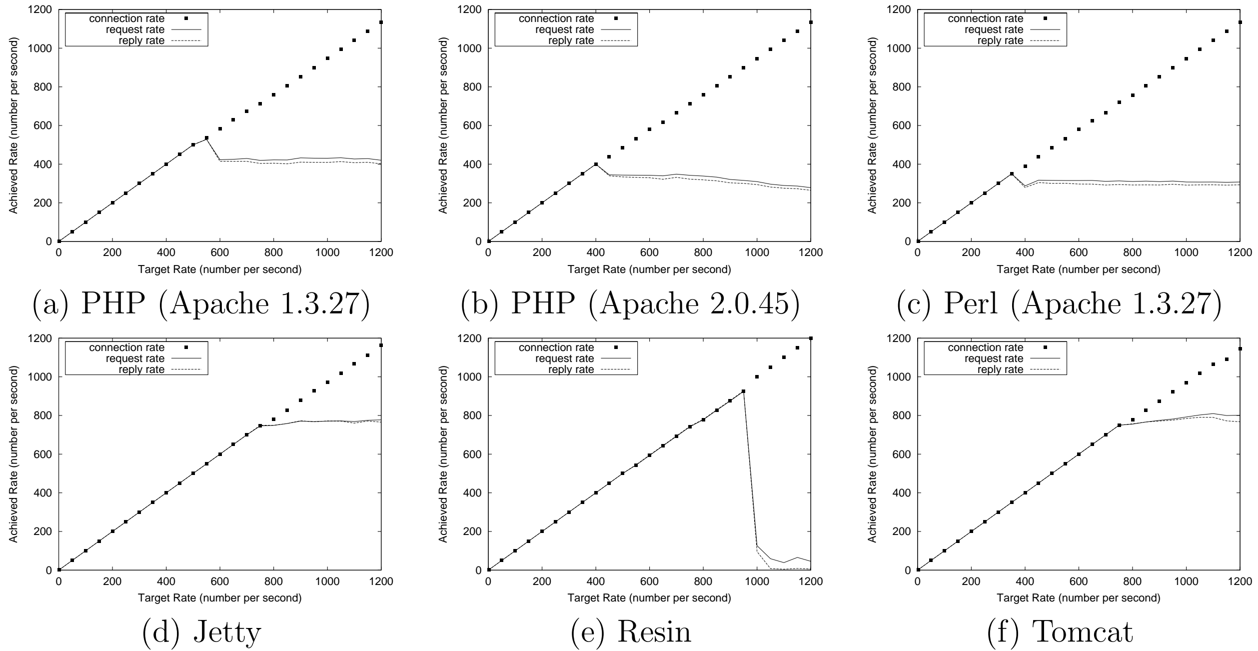
5.3 Dynamic Workloads (with database access)

In this section, we analyze the performance of the different dynamic content generation strategies for two different workloads. In these experiments, a single SQL INSERT and a single SQL SELECT command are executed when generating the response. The results for the 2 KB workload are given in Section 5.3.1. The results for the 64 KB workload appear in Section 5.3.2.

5.3.1 2 KB Dynamic Workload (with database access)

Figure 6 shows the results for the 2 KB dynamic responses requiring database access. Several observations can be made from these graphs. First, the three Java-based servers all significantly 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 ranges from 36% to 56% of the performance when no database access is required for dynamic content generation.

Among the three Java-based servers, Resin has the highest peak performance at 927 requests per second. However, Resin again behaves poorly under overload. Jetty and Tomcat peak near 750 requests per second. Both of these servers appear stable under overload conditions.



(a) PHP (Apache 1.3.27) (b) PHP (Apache 2.0.45) (c) Perl (Apache 1.3.27)

(d) Jetty (e) Resin (f) Tomcat

Figure 6: Experimental Results for 2 KB Dynamic Responses (with Database Access)

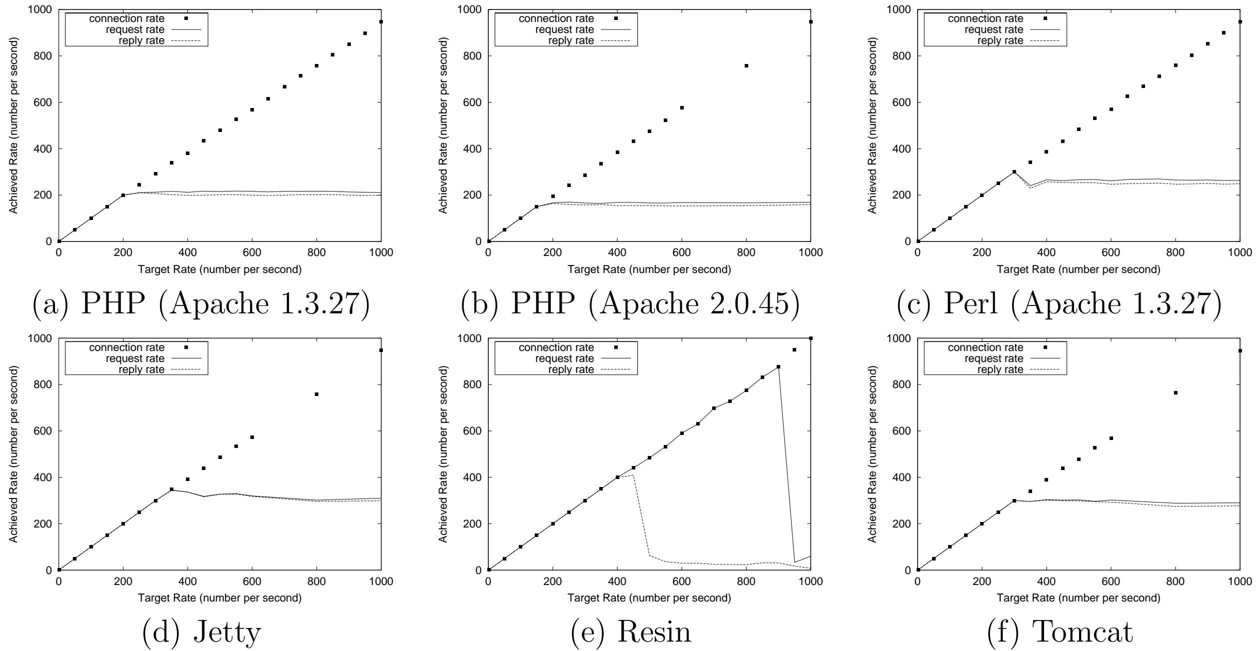
5.3.2 64 KB Dynamic Workload (with database access)

The results of the experiments for 64 KB dynamic responses requiring database access are shown in Figure 7. 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 or Perl. As noted in Section 5.2.2, Perl outperforms PHP for the 64 KB responses. Jetty and Tomcat have peak performance around 300 requests per second, and behave well under overload. Resin again achieves the highest peak performance (approximately 400 requests per second), but then sacrifices response rate and focuses on accepting requests. As a result, Resin accomplishes little useful work once in an overload condition.

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 dynamic Web content generation technologies.

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



(a) PHP (Apache 1.3.27) (b) PHP (Apache 2.0.45) (c) Perl (Apache 1.3.27)

(d) Jetty (e) Resin (f) Tomcat

Figure 7: Experimental Results for 64 KB Dynamic Responses (with Database Access)

generation itself. Our experiments have quantified each of these effects. Combined, these effects typically result in the halving (or worse) of the peak request rate supported by a server. Second, today's technologies for dynamic Web content generation offer several 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 well 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, Tomcat), and some are not (Resin). This observation suggests that consideration of overload behaviour may be just as important as the peak request rate supported when Web site administrators are choosing dynamic Web content generation technologies.

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.

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 Nayden Markatchev for his technical support in the installation and configuration of machines in the ELISA lab for these experiments.

References

[1] 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.

[2] Caucho Technology, “Resin Core page”, 2003, `http://www.caucho.com/resin/`.

[3] 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.

[4] 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.

[5] 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.

[6] Mort Bay Consulting, “Jetty Product Page”, 2003, `http://jetty.mortbay.org/jetty/`.

[7] 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.

[8] 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.

[9] 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.

[10] Netcraft Web Server Survey, `http://www.netcraft.com/survey`.

[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.200304/apachemods.html`.

[13] R. Simmonds, C. Williamson, R. Bradford, M. Arlitt, and B. Unger, “Web Server Benchmarking Using Parallel WAN Emulation”, `http://www.cpsc.ucalgary.ca/~carey/papers/iptne.pdf` (A short 2-page abstract of this paper appears in ACM SIGMETRICS 2002).

[14] Standard Performance Evaluation Corporation, `www.spec.org`

[15] Red Hat, “Tux Web Server Manuals”, `www.redhat.com/docs/manuals/tux`

[16] 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.

[17] 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.