Performance Comparison of Dynamic Web Platforms

Varsha Apte[1,*], Tony Hansen[2] and Paul Reeser[2]
[1]**Department of Computer Science and Engineering, Indian Institute of Technology -Bombay, Mumbai 400 076, INDIA**
[2]**AT&T Labs, 200 Laurel Ave, Middletown, NJ 07748, USA**

Keywords: Dynamic**,** Web, CGI, FastCGI, C++, Java, Servlets, JSP, Performance, Comparison

Over the last few years, the World Wide Web has transformed itself from a static content-distribution medium to an interactive, dynamic medium. The Web is now widely used as the presentation layer for a host of on-line services such as e-mail and address books, e-cards, e-calendar, shopping, banking, and stock trading. As a consequence, HTML files are now typically generated dynamically *after* the server receives the request. From the Web-site providers' point of view, dynamic generation of HTML pages implies a lesser understanding of the real capacity and performance of their Web servers. From the Web developers' point of view, dynamic content implies an additional technology decision: the Web programming technology to be employed in creating a Web-based service. Since the Web is inherently interactive, performance is a key requirement, and often demands careful analysis of the systems. In this paper, we compare four dynamic Web programming technologies from the point of view of performance. The comparison is based on testing and measurement of two cases: one is a case study of a real application that was deployed in an actual Web-based service; the other is a trivial application. The two cases provide us with an opportunity to compare the performance of these technologies at two ends of the spectrum in terms of complexity. Our focus in this paper is on how complex vs. simple applications perform when implemented using different Web programming technologies. The paper draws comparisons and insights based on this development and performance measurement effort.

1. INTRODUCTION

The World Wide Web (WWW) first emerged a decade ago as a medium to render hypertext documents that were stored on the Internet, on a user's computer, using special software (the browser) and a new protocol (HTTP: HyperText Transfer Protocol). For the first few years, the WWW grew primarily as a new medium in which static content could be published, and information shared. The content was published in the form of HTML (HyperText Markup Language) files, which were served by Web servers, on requests from browsers. However, over the last few years the WWW has transformed itself from a static content-distribution medium to an interactive, dynamic medium. Content on the Web is now often personalized, and therefore dynamically generated. The Web is now widely used as the presentation layer for a host of on-line services such as e-mail, e-cards, e-calendar, and address books, shopping, banking, and stock trading. As a consequence, the HTML files that are rendered by the client's browser are now typically generated dynamically *after* the Web server has processed the user's request.

---

[*] This work was done while this author was with the Network Design and Performance Analysis Department, AT&T Labs, Middletown, NJ 07748, USA. This author has previously published under her maiden name of *Varsha Mainkar*.

This dynamic generation of HTML files has not happened without an associated performance cost. Just when Internet users were getting accustomed to “click-and-wait” on dial-up lines due to graphics-rich Web sites, dynamically generated content started proliferating on the Web. Now users must wait not only for the network delay but also for the server-side *processing* delay associated with serving a request dynamically. In many cases, this is turning out to be the largest component of the delay. From the Web-site providers’ point of view, dynamic generation of HTML pages implies a lesser understanding of the real capacity of their Web servers. The vendor-provided “hits-per-second” capacity of the Web server is no longer enough, as this only pertains to static HTML files.

From the Web developers’ point of view, dynamic Web content implies an additional technology decision: the Web programming technology to be employed in creating a Web-based service or product. This decision is based on several factors. Among the factors considered are ease of programming, richness of features, maintainability, reliability, and performance. Since the Web is inherently interactive, performance is a key requirement, and often demands careful analysis of the systems.

In this paper, we compare the performance of four Web programming technologies, namely Java Servlets, Java Server Pages, CGI/C++ and FastCGI/C++ (a smaller version of this study was presented in [1]). The study was motivated by a real need to make a technology choice for developing software that would support a Web-based service. A study of existing literature showed varying conclusions about the performance superiority of one technology over the other. Proponents of the Java platform consistently claim superior performance of servlet technology mainly due to the elimination of the overhead of process creation [2, 3]. On the other hand, a study done in [4] that compares CGI, FastCGI and servlets showed servlets to be the slowest of the three. A similar comparative study has been done in [5], focusing on Web-to-database applications, which showed Java servlets to be better than CGI programs. Furthermore, although [4] briefly provides some explanation about the results, an in-depth analysis and insight about the results was not available in the literature that we surveyed. One conclusion from the variability of the results was also that the performance of the technology depends on the application.

In this paper, the comparison is based on two cases: one is a case study of a complex application that was deployed in an actual Web-based service; the other is a “trivial” application. The methodology of performance analysis was stress testing and measurement. Performance measurement (rather than

modeling) made most sense in this effort, since a quick turnaround of results was necessary and the accuracy of results was required to be high.

The two cases (i.e., the complex and the trivial) provided us with an opportunity to compare performance of these technologies at two ends of the spectrum of applications, in terms of complexity. The performance "order" of different technologies is seldom absolute – it depends greatly on the nature of the application. Our focus in this paper is on how complex vs. simple applications perform when implemented using different Web programming technologies. Among the papers surveyed, we believe this is the only paper presenting a systematic bottleneck analysis of each of the technologies, and the only one that demonstrates that the performance "ranking" could be reversed, based on the application, while providing insight on why this happens.

The main observations from this work were as follows: In general, FastCGI outperformed CGI, while JSP outperformed Java servlets. In the case of a complex application, the CGI-based technologies outperformed the Java-based technologies by a factor of 3-4x, with Java performance limited by software bottlenecks in the JVM. In the case of a trivial application, the relative performance was reversed, as Java outperformed CGI by a factor of 2-3x.

The rest of the paper is organized as follows: in Section 2 we provide the motivation for conducting such a comparative study, and in Section 3 we describe briefly the technologies that we compared. Section 4 describes the performance measurement and analysis methodology, Section 5 describes the case-study testing results in detail, and Section 6 describes the results of testing a trivial application. Finally, Section 7 summarizes our results, and Section 8 provides some concluding remarks.

2. MOTIVATION

The application context for this case study was a new Web-based messaging service. Such a service would involve a back-end that would include the core messaging-related servers (IMAP, POP, LDAP, SMTP), and a front-end, or a presentation layer that would serve as middleware between the Web browser and the messaging server. The core servers were chosen to be established messaging products. The presentation layer would consist of a Web server (off-the-shelf) and a new dynamic page generation engine to be developed, which would

- Read data sent by the user, through the HTTP protocol

- Do the necessary protocol conversions (to IMAP, LDAP, …)
- Carry out the action encoded in the HTTP request, using the appropriate protocol with the back-end servers
- Format the result as a Web-page, and
- Return the result to the user's Web browser through the Web-server

One way to write such a program that generates Web pages dynamically is to code the logic, and then embed "print" statements in the program that write out static HTML code, along with printing other string variables that have been populated dynamically. A better and more popular way is using *templates.* The middleware we studied used a template-based design for producing dynamic Web pages. This method involves writing Web page templates which have static HTML code that specify the design of the Web page, interspersed with *scripting tags* that are interpreted by a *template server* program. These tags instruct the template server program on the action that is to be taken (e.g. populating a field with the subject line of a message) on reading the tag. The template server program, therefore,

- Parses the specific template accessed by a user request
- Interprets the tags and communicates with the back-end servers
- Populates the fields that are to by dynamically populated

The result is a dynamically generated Web page. The architecture of such a system is shown in Figure 1. Given the accelerated time-to-market goals and limited development time, the natural choice of technology for the template server was the one perceived to be powerful, feature-rich and yet easy to use and deploy – namely, Java servlets.

An initial effort was done to quantify the performance of this page generation engine, to see whether it could meet the expected performance requirements. We conducted a series of stress tests using a commercial load driver to generate repeated requests to the servlet engine at various levels of concurrency (simulated users). The test configuration consisted of a Windows NT server running the load generation scripts (driver), a Solaris server running the front-end software, and a Solaris server running the back-end application (for this test, a POP3/IMAP4 mail server and an LDAP directory server). Hardware (CPU, memory, disk, I/O) and software resource consumptions were measured on all machines. In addition, end-to-end user-perceived response times were measured.

The driver scripts emulated a prescribed number of concurrent users repeatedly generating the same request (e.g., read message, send message, etc.). The number of concurrent simulated users was varied from 1 to 20. The number of repeated requests per user at each concurrency level (typically 2000) was sufficient to achieve statistical stability. The tests were run in "stress" mode; that is, as soon as a user receives a response, it immediately submits the next request (i.e., with negligible client delay). Each of the N simulated users does so independently and in parallel. As a result, the concurrency level (i.e., the number of requests in the system) equals N at all times.

The results of the stress tests for a particular request type (read a 20KB message) are shown in Figures 2 and 3. In particular, Figure 2 plots the end-to-end response time (normalized) on the left-hand axis, and the front-end CPU utilization on the right-hand axis, as a function of the concurrency level. As can be seen, the response time curve begins to ride along a linear asymptote (shown by the dotted line) after only 7 concurrent users. That is, response time increases proportionally with the number of users, indicating saturation in a closed system [6]. Additionally, CPU utilization levels off after 11 users at 65-70% (indicating a non-CPU system bottleneck).

Equivalently, Figure 3 plots end-to-end response time as a function of throughput (requests/sec). As can be seen, the maximum system throughput peaks at about 2 requests/sec, and then degrades under overload to about 1½ requests/sec. In other words, there was actually a drop in capacity of 25% (a "concurrency penalty"), likely due to context switching, object/thread management, garbage collection, etc.

A sizing analysis based on the expected customer growth and usage behavior, together with these initial capacity results, suggested that the resulting hardware costs would be prohibitively large. It was also clear that the *scalability* of this application was poor. The resource consumption results demonstrated that the application could not make full use of the resources available to it (especially CPU).

The first question to be answered was whether or not the application was implemented in the most optimal manner. The initial phase of the performance enhancement effort was in this direction. Several key optimizations were performed on the Java servlet code, and several critical bottlenecks were discovered and resolved in the end-to-end architecture (described in a separate paper [6]). The resulting improvement in the read request throughput is shown in Figure 4.

Sizing analysis on the improved and optimized code showed that the resource requirements were still quite substantial, and the application still hit a premature wall (in terms of scalability). The CPU continued to level off, now at about 90%. As a consequence to this analysis, an additional effort was launched to re-assess the choice of the dynamic technology itself. The plan was to analyze applications that had the same functionality, but were implemented in different languages or technologies, within identical environments. The technologies chosen, including the initial version, were Java servlets, Java Server Pages (JSP), Common Gateway Interface (CGI) with C++ programs, and Fast CGI with C++ programs. The following section gives a brief introduction to these Web-programming technologies.

3. DYNAMIC WEB PLATFORMS

There are myriad different technologies employed to produce HTML files dynamically, after the arrival of an HTTP request. CGI, FastCGI, Java Servlets, JSP, Active Server Pages (Windows NT only), PHP Hypertext Preprocessor, JavaBeans, and Enterprise JavaBeans are among the many technologies currently used for dynamic processing of Web requests. In this effort we focus on four technologies: CGI, FastCGI, Java Servlets and JSP. The following subsections give a brief overview of these technologies.

3.1. CGI

Common Gateway Interface (CGI) [7] was the earliest technology used to service an HTTP request by a program, rather than by sending a static file. In this technology, a program is written in any language (C, C++, Perl, shell, etc.) for processing a user request and generating output in a browser-viewable format (HTML, GIF, etc.). When a request arrives with a URL pointing to the CGI program, the Web server creates a separate process running that particular CGI program. The Web server sets various environment variables with request information (remote host, HTTP headers, etc.), and sends user input (form variables) to the CGI process over standard input. The CGI process writes its output to standard output. When the processing is complete, the CGI process exits, and the Web server sends the output to the browser. When the next request comes in, all of these steps are repeated.

CGI performance is affected greatly by the overhead of process creation every time a dynamic request is served. In Unix, the operating system creates the CGI process by first cloning the Web server process (a large process) and then starting the CGI program in the address space of that clone. This requires a lot of system resources and is the main reason for performance problems in this technology.

3.2. Fast CGI

FastCGI [8] addresses the process creation problem in CGI by having processes that are *persistent*. In this technology, the Web server creates a process for serving dynamic requests either at startup or after the first dynamic request arrives. After creation, this process waits for a request from the Web server. When the Web server gets a request that requires dynamic processing, it creates a connection to the FastCGI process (over a pipe, if local, or TCP/IP if on a remote machine), and sends all the request information on this connection to the process. After processing is done, the FastCGI process sends output back on the same connection to the Web server (using a Web server API), which in turn forwards it to the client browser. The Web server then closes the connection, and the FastCGI process goes back to waiting for a new connection from the Web server. Thus, FastCGI essentially eliminates the overhead of process creation, potentially improving CGI performance greatly.

3.3. Java Servlets

Java was first introduced as a technology for sending platform-independent code from a Web server to a Web browser, where these “Java applets” would run inside an environment called the Java Virtual Machine (JVM). As Java grew in popularity, the scope of Java grew to fit the demand. “Server-side Java applets” – Java servlets [2] – were introduced, allowing developers to use Java to write programs on the server that process dynamic requests. The main advantage of these servlets was the use of Java’s natural multi-threading architecture. When a request comes in, a new Java *thread* is created to execute the servlet program. The servlet accesses user information using the servlet API, processes it and sends the output back to the Web server, using a special API to the Web server. The Web server returns this output to the browser. Servlets are quite popular because they are based in Java technology, and offer all of the features that make Java programming so popular. Initially, they were also claimed to have solved the performance problems of CGI, since thread creation is much more lightweight than process creation.

3.4. Java Server Pages

Java Server Pages (JSP) [9] is a technology that allows programmers to cleanly separate the presentation part of an HTML file from the information part, which is created dynamically. A JSP file contains standard HTML code, interspersed with the JSP code that specifies how to “fill in” the “placeholders” for the dynamic code. This separation of functionality addresses a difficulty faced in servlet

or CGI development, where the programs had to return the dynamically generated content in HTML format so as to be displayed by the Web browser. That is, the HTML formatting was part of the servlet code, and any changes to the HTML design meant changing the servlet code itself. Developers solved this problem by writing HTML template files with “tags” (the technique used in this application) that are processed by the servlet, and replaced with dynamically generated information. However, this implied that each servlet development team came up with its own custom tags. Now, JSP offers a de-facto standard for doing just that. A JSP file is compiled into a servlet, so in the end it uses Java servlet technology, but with a different programming interface.

4. TESTING ENVIRONMENT & METHODOLOGY

As described in Section 2, the messaging application was originally built using a page generation engine programmed in Java servlets. The page generation engine was a specific case of software that can be termed as *template server programs.* Template server programs use Web templates to specify the look and feel of a Web page, and employ *tags* that are read and interpreted by the server program. For our performance comparison effort, we chose other such template server programs.

For the CGI case, an existing C++ CGI template server program was used, that implemented the same functionality. The relevant changes (optimizations) were made to the original servlet implementation to bring the differences down to the essential technology differences. The FastCGI implementation was done specifically for the testing, by minimally changing the CGI C++ code.

JSP is essentially a standardized Java template server technology. A new JSP implementation was done for the purpose of testing.

The environment used to do the performance tests was as follows:

- Server hardware: 2x360Mhz Sun Ultra-60, with 0.5 GB memory.
- Server software: Solaris 2.6, JDK 1.2.1, Jrun 2.3.3 servlet engine.
- Web servers: Netscape 4.1 and Apache 1.3.12[1].

[1] For implementation reasons, we used two different Web servers in the tests (FastCGI worked only with the Apache server). The CGI tests used the Apache server, and the Java tests used the Netscape server. Sanity checks were done to confirm that this variation would not affect the major conclusions of this study. Specifically, the CGI implementation was measured in both the Netscape and Apache environments, and did not show major differences that would affect any conclusions derived in this paper.

- Load generation client: PC with Windows NT.
- Load generation software: Silk Performer™ 3.6.
- Network: Gigabit Ethernet.

The setup of each of the tests was as follows:

- All servers were stopped and re-started before each test.
- Each virtual user (programmed in Silk Performer) does a typical messaging session where the user logs in, lists messages, reads some messages of varying sizes and then logs out. There are no think times between the transactions issued by the user. As soon as the user process gets a response back, it issues the next transaction.

The test started with one user and added a user every 10 minutes, increasing up to 30 users (enough to stress the fastest of our implementations).

4.1. Performance Measurement

Measurements were taken at both the client and the server. Client measurements were recorded using Silk Performer, while server measurements were recorded using standard Unix measurement tools (*sar, netstat, mpstat, ps)* [10].

The throughput (in sessions/second) at the client was calculated in the Silk Performer user script by counting the total number of sessions completed by all users in the 10-minute window during which the number of users remained the same, and dividing this number by 600 seconds. The response times (to complete one session) were recorded and an average in the 10-minute window was calculated.

On the server side, 10-second snapshots of server measurements were taken (CPU, memory), and were used to produce the 10-minute averages. (Taking snapshots on a smaller scale allowed us to see transient states of the server resources.)

4.2. Converting “Stress Tests” to “Load Tests”

Using the averaged measurements in the 10-minute windows gives us the measurements corresponding to increasing numbers of users: i.e. response time with 1, 2, …, 30 users, or CPU utilization with 1, 2, …, 30 users. However, just knowing how many virtual users were generating load on the server does not necessarily tell us the actual load on the server in its correct unit: session requests/second (e.g. see Figures 2 and 3). For example, if a system has reached its saturation point, adding users may not

proportionally increase the session setup rate at the server (or even at all). This is because the users are essentially slowed down by the system response time, and can only generate load as fast as the system capacity permits [6]. It is important, therefore, to derive the real metric of load: the session requests sent to the server per unit time, and plot performance measures against this metric.

In our case, the Silk Performer script was programmed to record the number of sessions completed during a 10-minute window corresponding to a certain number of users, therefore we had direct measurements of the sessions/second (i.e. the throughput).

In the plots depicted in the following section, each sample point (average response time, CPU utilization, page faults, etc.) corresponds to the average measurement from the 10-minute window corresponding to a certain number of users. However, the unit of the X-axis is the corresponding throughput (in sessions/second), *not* the number of users. Readers should keep this in mind while studying the graphs, as some of the sample points are clustered together. This clustering occurs because after system saturation, although the number of users can be increased, the throughput does not increase (it may, in fact, decrease).

4.3. Analysis approach

The purpose of this study was not only to understand which implementation performed best, but also to pinpoint the bottleneck resources for each technology, and to gain insight into why one technology out-performed the other. For each implementation, we carried out the following steps:

- Determined the maximum achievable throughput using stress tests
- If possible, found the bottleneck resource, and verified that the resource capacity was equal to the maximum throughput achieved by that implementation.
- Where the exact bottleneck resource was not identifiable, we eliminated resources that could not be bottlenecks.

We also used the two applications to understand the effect of the complexity of the applications. For both the cases, the above steps were done. The measurements we took were CPU time, and page fault rate, to monitor the usage of the processor and memory resource, respectively. To gain further insight, we also studied the user time and system time used by each implementation, separately. The results and insights are described in the next section.

5. RESULTS AND COMPARISON – MESSAGING APPLICATION

The main metric used to compare the technologies was the maximum achievable throughput, gated by a response time criterion. Figure 5 shows the average session response time vs. session throughput. There are two observations one may make from this chart. First, the throughput thresholds up to which the system is essentially stable, with response times within "reasonable bounds" and increasing gracefully are as shown in Table 1. Above these thresholds, response times increase sharply, indicating that the system has become unstable. Second, given a performance requirement of 10 seconds average session response time, the throughput at which this requirement can be met, for each case is as shown in Table 2.

In this case it turns out that the thresholds at which the systems become unstable are reached before the response time criterion is violated. The capacity of the system should be determined by the minimum of these two, so Table 1 shows the capacity of each of the technologies, in terms of maximum achievable throughput. FastCGI/C++ is the clear winner in this metric, followed by CGI/C++, JSP and Java servlets, in that order. The following analysis tries to understand why this happens and what the bottlenecks encountered by each technology are.

5.1. Bottleneck Analysis

Several system resources are potential bottlenecks: CPU, memory, network, back-end systems, the load generator, software resources such as thread limits, buffer sizes, and Java-specific resources, etc. In the following sections we examine the possibility of each of these resources being the bottleneck.

CPU

We start with a look at the CPU. Figure 6 plots CPU utilization vs. throughput for each case. CPU utilization is calculated as a sum of the time the CPU spends in user mode and system mode (indicated as "usr+sys"). We see from the chart that both the CGI and FastCGI implementations were able to make full use of the CPU. However, both the JSP and Java servlet implementations use up to about 90% of the CPU, after which the throughput starts dropping and the CPU utilization starts dropping, although more users are being added. This drop alone is not a symptom of a problem (CPU utilization should always increase and decrease with throughput). However if you observe Figure 5 again, you can see that at the points where CPU utilization stops increasing, the response times continue increasing sharply, as more users are added. This leads to the conclusion that the Java technologies reach a non-CPU bottleneck.

Some insight into use of the CPU resource can be gained by calculating the actual CPU time used by each session. If ρ is the utilization and λ is the throughput, then this time is given by $\tau = \rho/\lambda$ [11]. Figure 7 shows τ (in ms) used by each session, plotted against the throughput. This is an interesting chart; it shows quite a steady plot for the CGI/FastCGI implementations, but a highly variable one for the Java technologies. Also, asymptotically, the Java implementations show a trend of increasing CPU time per session, as the number of users increases, suggesting that execution time in the Java implementations is greatly influenced by factors such as the size of the Java heap.

The average CPU time per session for all implementations is shown in Table 3 (along with the factor by which this is worse than the fastest implementation). In the case of the CGI/C++ implementation, the average value of τ is 981 ms. Since there are 2 CPUs in the server machine, the overall CPU throughput is 2 × 1000/981 = 2.04 sessions/second, which corresponds almost exactly to the maximum throughput as determined in Table 1. Thus, the CPU is the bottleneck resource in the CGI implementation.

A similar analysis for FastCGI shows CPU throughput to be 2 × 1000/765 = 2.61 sessions/second, which again corresponds to the maximum throughput for FastCGI as determined in Table 1. Thus, the CPU is the bottleneck resource for the FastCGI implementation.

The average CPU time per session for the servlet implementation is 2468 ms, which implies that the CPU throughput should be 0.8 sessions/second. This is higher than the achieved throughput of 0.6 sessions/second, and further confirms that the servlet implementation runs into a non-CPU bottleneck. Similarly, the CPU throughput of the JSP implementation is 0.93, which again is higher than the achieved throughput of 0.8 sessions/second.

A further analysis of how the CPU times are actually used is shown in Table 4. This table reaffirms the effects of the characteristics of each technology: most of the time in the CGI implementation is spent in creating the CGI process, which is done in system mode. Almost all of the improvement that FastCGI does is on this time (reducing it to 337ms from 500 ms). However, it still remains substantial, possibly due to the setting up and tearing down of connections, and due to some initial creation of Apache Web server processes (until they reach a certain maximum). The Java implementations also improve substantially on the system time (reducing it to 173 ms from 500 ms). However, the time spent in the user mode by these implementations is much higher (e.g. 2295 ms vs 428 ms).

Memory

Since CPU was not the bottleneck in the Java implementations, we turned our attention to memory bottlenecks. A measure of memory problems is paging activity; specifically, page faults. Page faults occur either when multiple processes sharing a page try to write to it, in which case a copy of the page must be made, or when needed pages have been reclaimed by the operating system. Page fault activity increases when a process size becomes too large, when a file is read into memory, or when a process is started, since this involves bringing code into memory. Figure 8 shows page faults/second vs. sessions/second. The chart shows that in fact it is the CGI implementation that has the highest indicators of paging activity. (Note that the Y-axis has a logarithmic scale). The paging activity for the FastCGI and the Java technologies is an order of magnitude lower than that of CGI. The high paging activity in the CGI implementation is clearly due to the process creation that occurs whenever a dynamic page generation request arrives. The initial increase in paging activity in the FastCGI case is due to the creation of Apache Web server processes. The activity levels off once the specified maximum number of processes has been created. Since the CGI and FastCGI implementations showed higher throughputs, despite having higher paging activity, we ruled out paging as the bottleneck in the Java implementations.

Other resources

For the CGI/FastCGI C++ implementations, the CPU is the clear bottleneck. In the case of the Java implementations, we can rule out the capacity of the back-end mail and directory servers, the load generation system, and the LAN as the bottleneck, since a higher throughput was achieved by the CGI technologies. This analysis points to only one possibility, i.e. that of a software bottleneck. Since the Java implementations were highly optimized, we concluded that these were not due to poor implementation, but rather to bottlenecks inherent in the Java technology.

6. RESULTS AND COMPARISON – TRIVIAL APPLICATION

The analysis in Section 5 showed that time spent in the Java servlets was mainly in user mode, and although this technology improves substantially on the process creation overhead, it lost that benefit due to the poor performance of the actual application code. This was most likely because the application being studied was a complex one, involving parsing text, connecting to remote machines, encrypting/decrypting (of session cookies), etc. It is fair, therefore, to compare these technologies in another scenario, where

the actual processing done was simple. As an extreme case, we carried out a set of performance tests in the case where each of the dynamic technologies was used trivially, to simply return a static file.

Figure 9 shows a chart of response time vs. throughput for two technologies on which tests were done: Java servlet, and C++ CGI. A session in this test includes the same transactions as in the earlier test, except that the files returned as a result of the transaction are static HTML files.

Table 5 shows the maximum achieved throughput under stable conditions by each of the technologies. This table shows a very interesting reversal of order of performance compared to Section 5[2]. In fact, it confirms the expectation that if processing is not dominated by user-mode processing, and is simple enough that it does not expose underlying Java bottlenecks, then Java servlets can be very fast.

Figure 10 shows that in this case, it is the C++ CGI implementation that runs into a non-CPU bottleneck (CPU utilization peaks at 93%, then flattens out, even as throughput degrades). The Java servlet in this case drives the CPU to its (almost) maximum capacity. Figure 11 also shows the CPU time per session vs. throughput. As the throughput rate increases, the CPU time per session used by the CGI implementation increases, whereas the Java servlet CPU time per session levels off.

Table 6 shows the breakdown of the CPU time into user mode and system mode computation. The CGI-C++ implementation spends an average of 80% of its processing time in the system mode. The expectation is that this is due to the excessive amount of process creation that this implementation needs to do. Figure 12 confirms this: the page fault rate of the CGI implementation is two orders of magnitude larger than that of the servlet implementation. Clearly, the system is *thrashing* – it is spending all its time in paging activity while spending lesser time doing useful work, resulting in the throughput dropping sharply.

7. SUMMARY AND INSIGHTS

The two suites of tests that were carried out provided important insights into the performance behavior of different Web programming technologies. In this section we attempt to intuitively explain why the order of performance seen was the way it was in the testing of the messaging application:

[2] Note that the tests in this section were carried out under a somewhat different configuration than the messaging application tests (Netscape 3.6, instead of Netscape 4.1 and Apache Web servers). Therefore, absolute results from these tests (such as throughput, CPU milliseconds, page faults, etc.) should not be compared with the corresponding tests in Section 5. We only compare tests within these sections and the relative order of performance of the technologies across the two types of tests.

- Java servlets were the worst performing because the implementation and the technology involved essentially three steps of interpretation of a program at different levels:
    1. The proprietary scripting language that was “interpreted” by the Java servlet.
    2. The Java servlet that was interpreted by (or at least ran inside) a Java Virtual Machine.
    3. The Java Virtual Machine that itself ran on the host CPU.

  The existence of these three layers contributes to the large amount of CPU used. Additionally, the Java Virtual Machine layer contributes to the non-CPU bottlenecks observed in the analysis.
- Java Server Pages improved on Java servlets because it took out the first of the three steps listed above. In the JSP implementation, a file with the HTML script, and the Java code that constitutes the dynamic part of the file, are compiled as a whole into one Java servlet. Thus, when a request arrives, the servlet runs, fetching the necessary data from the back-end systems, but it no longer interprets a template file. This reduces the CPU overhead, which results in JSP performance being better than the servlet/template combination. However, since JSP is a Java technology, it eventually runs into the same software bottlenecks at the virtual machine level that the servlet implementation encounters.
- The CGI/C++ implementation basically does steps 1 and 3 of the steps outlined above (in a different way). It is native code interpreting a tag-based template file, which produces an HTML file as a result. Not having to go through the 2nd layer reduced the CPU overhead greatly. Additionally, there are no implicit software resources or mechanisms that become bottlenecks. Thus, this implementation was able to use the CPU more efficiently and more fully (driving it to its maximum capacity). However, the CPU spends a lot of time in system mode doing work related to creation of the CGI processes.
- FastCGI eliminates the CGI overhead of recurrent process creation, and consequently reduces the paging overhead. The CPU overhead is reduced, and FastCGI improves performance over CGI. As in CGI, FastCGI is unencumbered by any of the software bottlenecks that are characteristic of the Java technology. Therefore, FastCGI is the best performing of the technologies compared.

In summary, the reason why the performance order was reversed in the trivial application test was that the actual processing was much smaller compared to the overhead associated with each technology. Since the Java servlet technology has the least overhead (creating a thread instead of a process), it was the best performing in this test.

8. CONCLUSION

We used stress testing and measurement methodology to determine the performance pros and cons of using a particular technology to implement a template server. The conclusion from this study would be that FastCGI/C++ is the best choice for a dynamic Web platform when performance is the primary concern. This is because most applications supporting a major service will likely be complex, thus exposing the software bottlenecks inherent in Java technology. On the other hand this study showed that if the Web programs are simple, then the performance penalties that Java imposes are small, and JSP may be the right choice.

In reality, in most of the Web-based services, all priorities are subject to an overall cost-benefit analysis, not focused just on performance. Product management teams may set performance requirements initially; however, if a Java-based technology offers less development time, fewer problems to system integration and deployment teams, and wider support, then the performance requirements may be relaxed, and Java could be the choice that meets most needs adequately. Further, Java technology is constantly improving, and new measurement studies done on newer versions of the technology could produce different results. In any scenario, however, comparative performance measurement studies are always called for - this work was illustrative of how such a study can be done using stress testing, measurement, and bottleneck analysis.

ACKNOWLEDGEMENTS

The authors thank Mehdi Hosseini-Nasab, Wai-Sum Lai, and Patricia Wirth for their helpful comments.

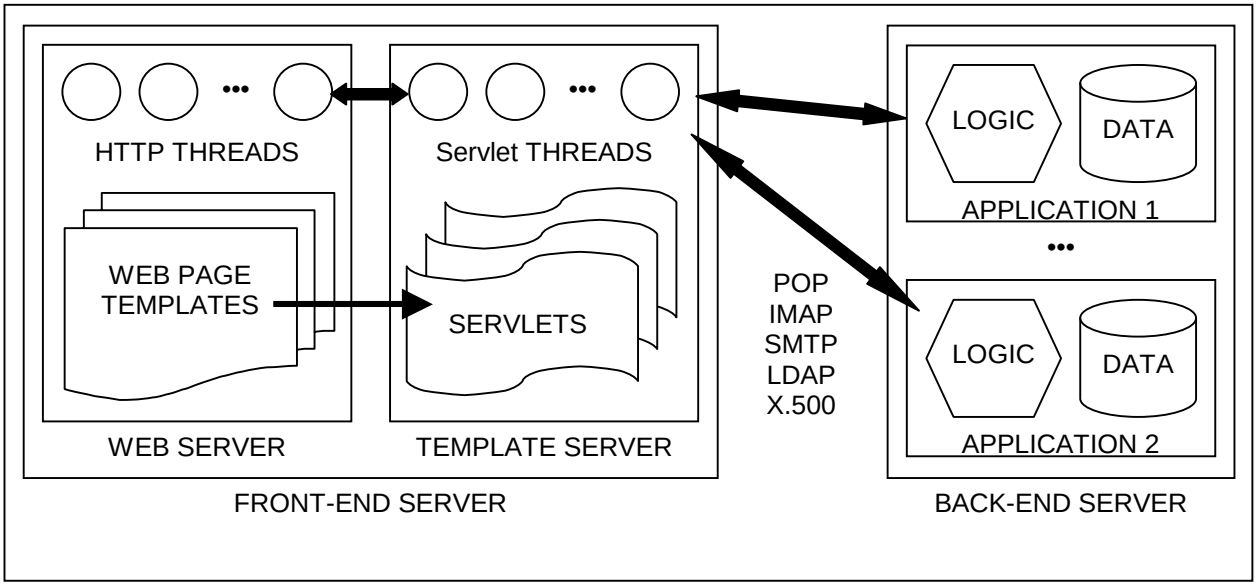

Figure 1: Template server architecture

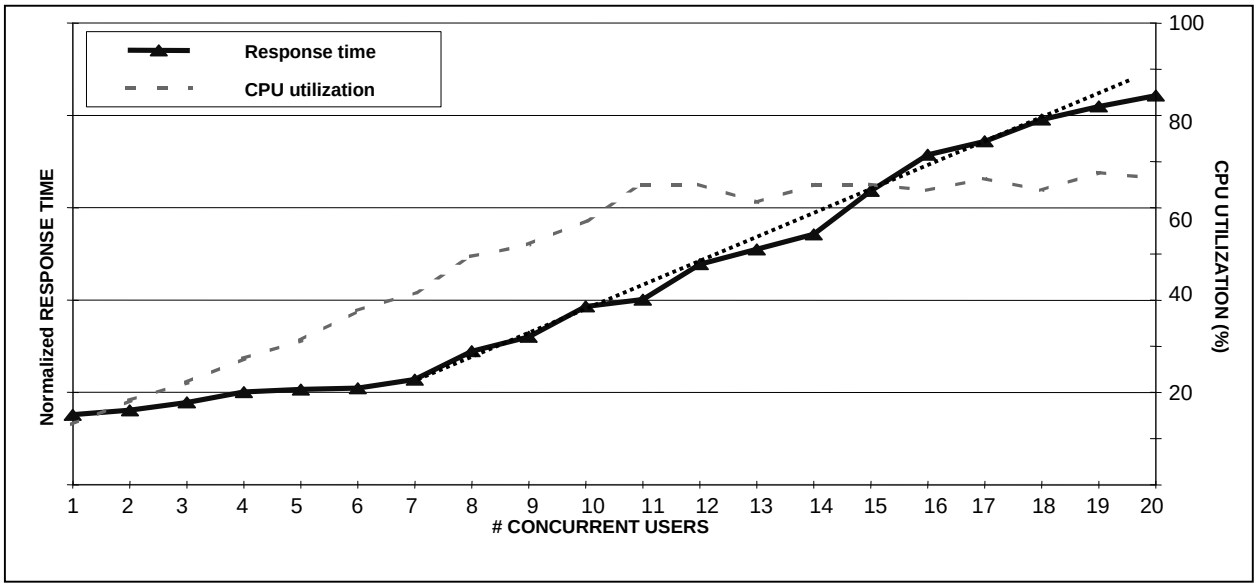

Figure 2: Results of initial stress tests of template server, plotted vs concurrent users

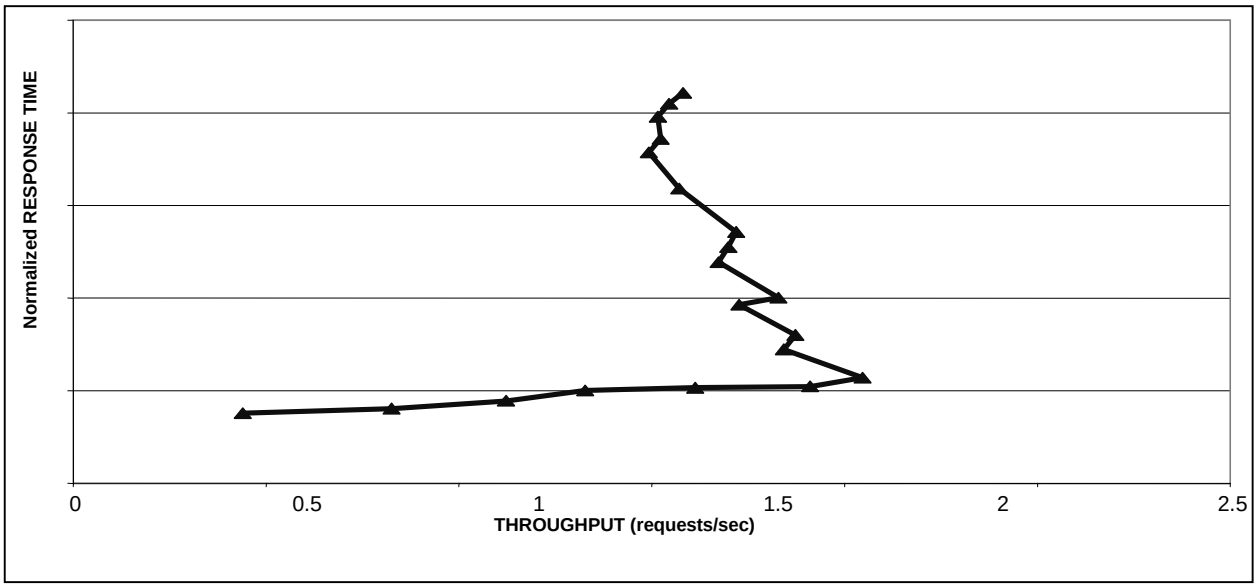

Figure 3: Equivalent load test results plotted vs requests/sec, from stress tests

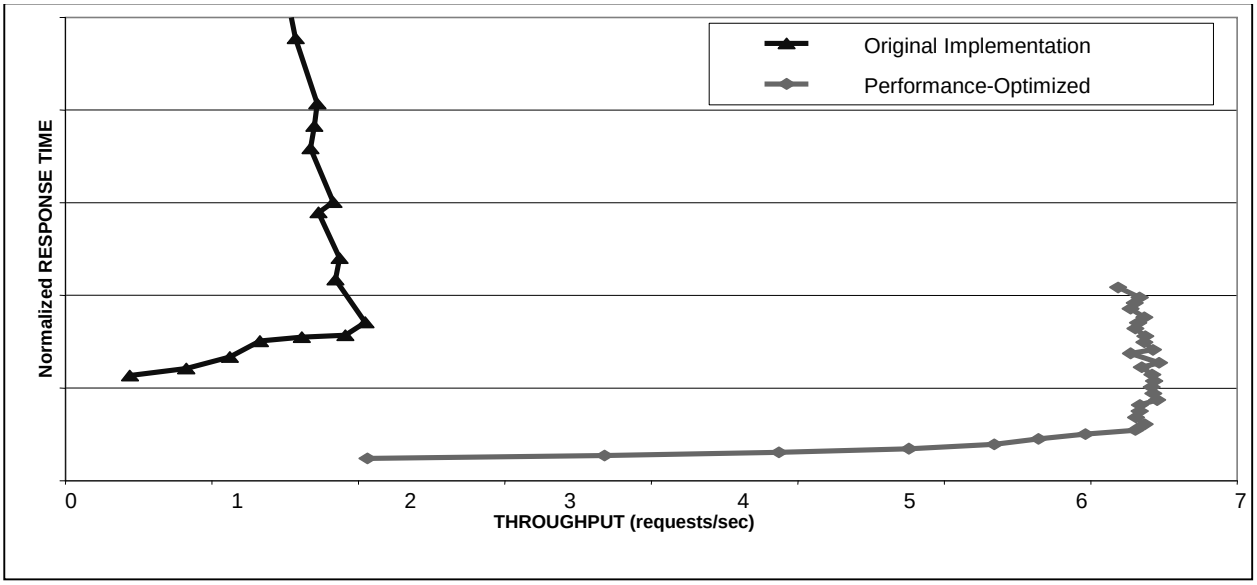

Figure 4: Results after optimization

Technology	Throughput
Java servlet	0.6 sessions/second
JSP	0.8 sessions/second
CGI	2.0 sessions/second
Fast CGI	2.5 sessions/second

Table 1: Maximum throughput under stable condition – template server

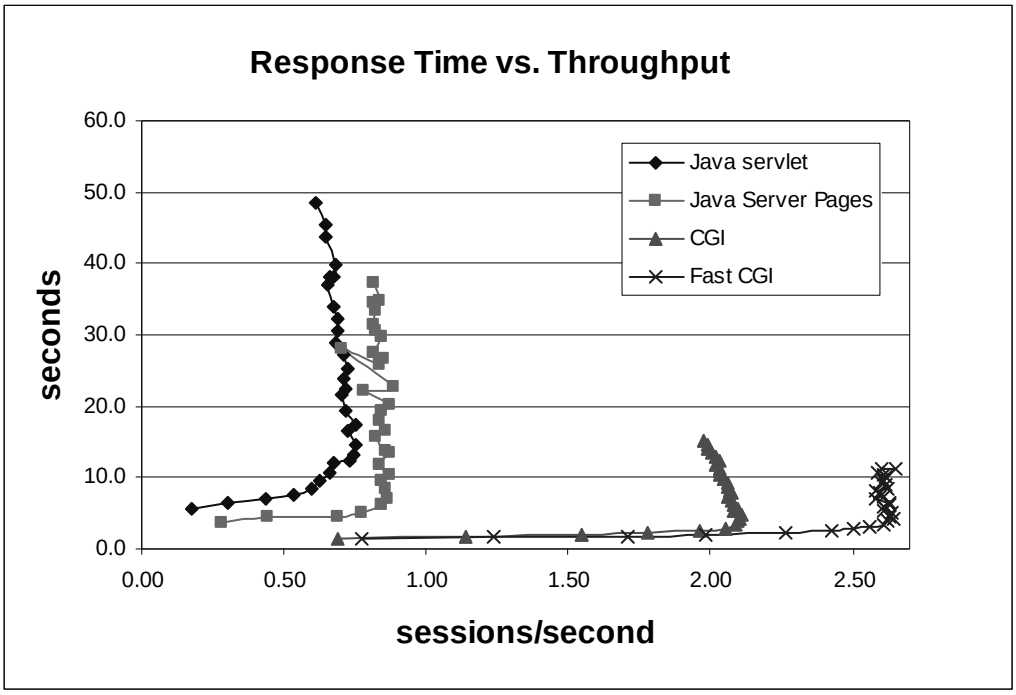

Figure 5: Response time vs. throughput – template server

Technology	Throughput
Java servlet	0.66 sessions/second
JSP	0.86 sessions/second
CGI	2.1 sessions/second
Fast CGI	2.6 sessions/second

Table 2: Maximum throughput gated by response time requirement – template server

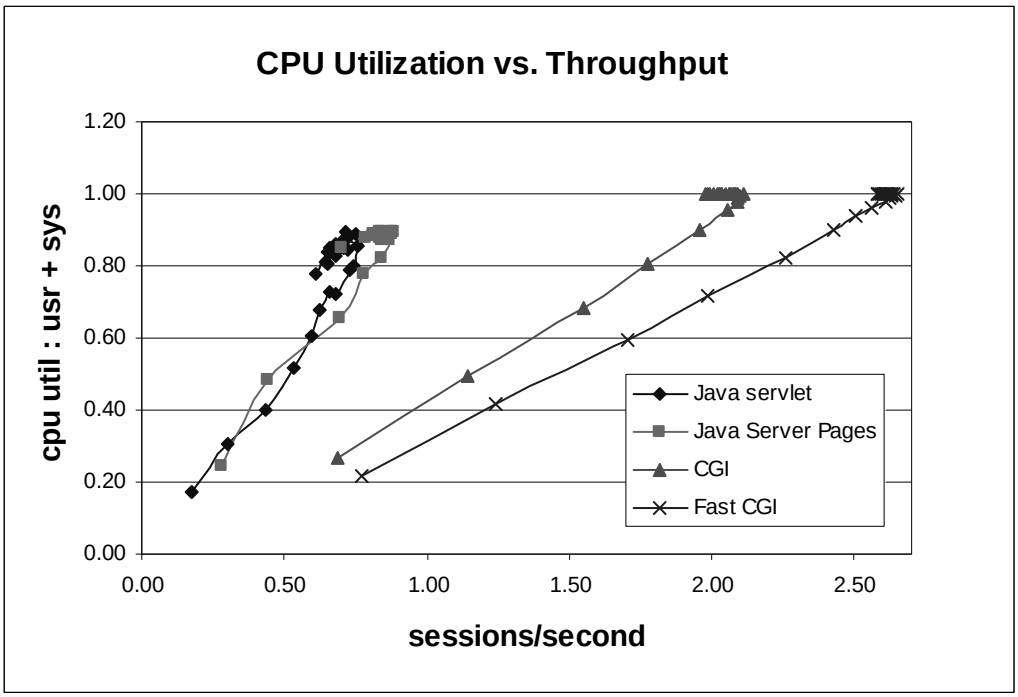

Figure 6: CPU utilization vs. throughput – template server

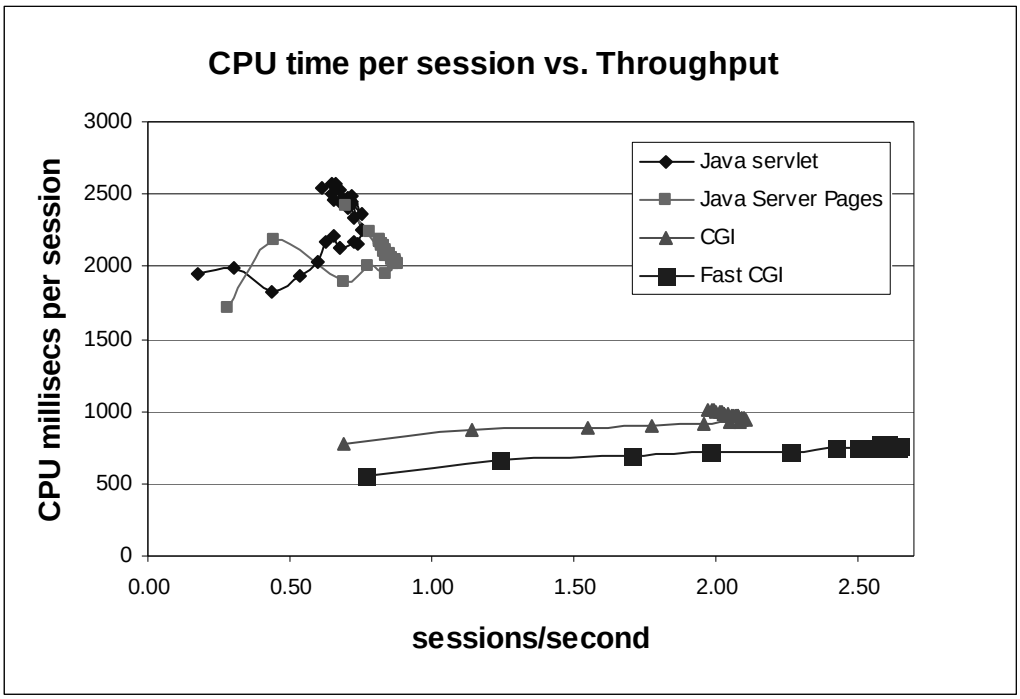

Figure 7: CPU ms per session – template server

Technology	Average CPU ms per session (t)	Factor higher than FastCGI
Java servlet	2468 ms	3.2
JSP	2131 ms	2.8
CGI	981 ms	1.3
Fast CGI	765 ms	1.0

Table 3: Average CPU time per session – template server

Technology	Time spent in system mode as % of overall CPU utilization	Actual ms spent in system mode	Actual ms spent in user mode
Java servlet	7 %	173	2295
JSP	9 %	192	1939
CGI	51 %	500	481
Fast CGI	44 %	337	428

Table 4: Breakdown of CPU milliseconds by user vs. system mode – template server

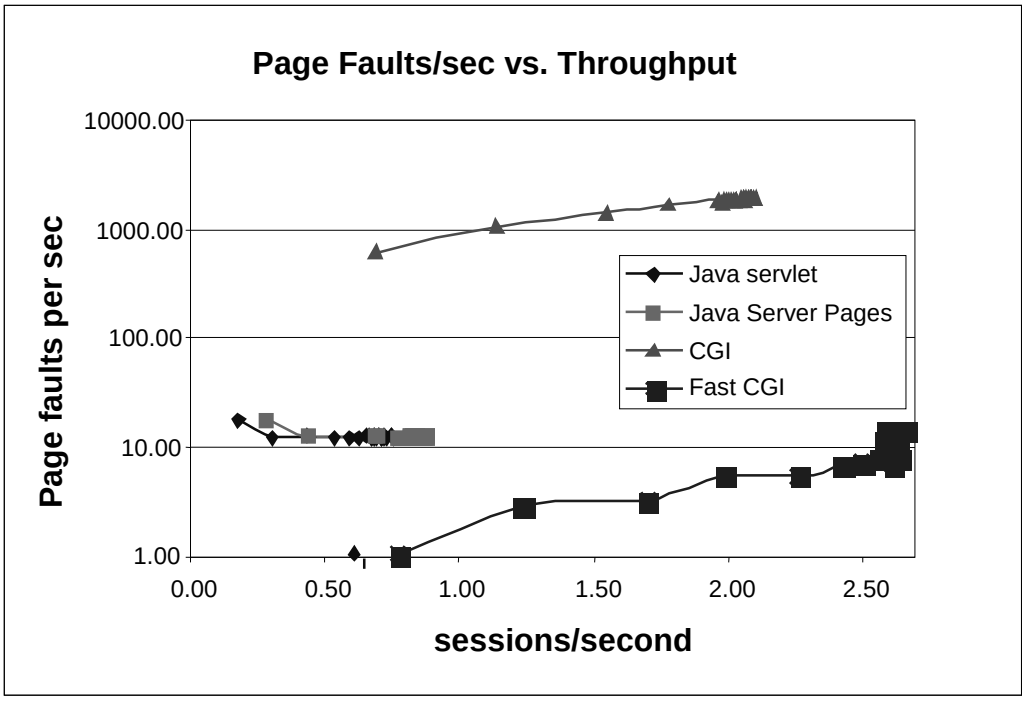

Figure 8*:* Paging activity comparison – template server

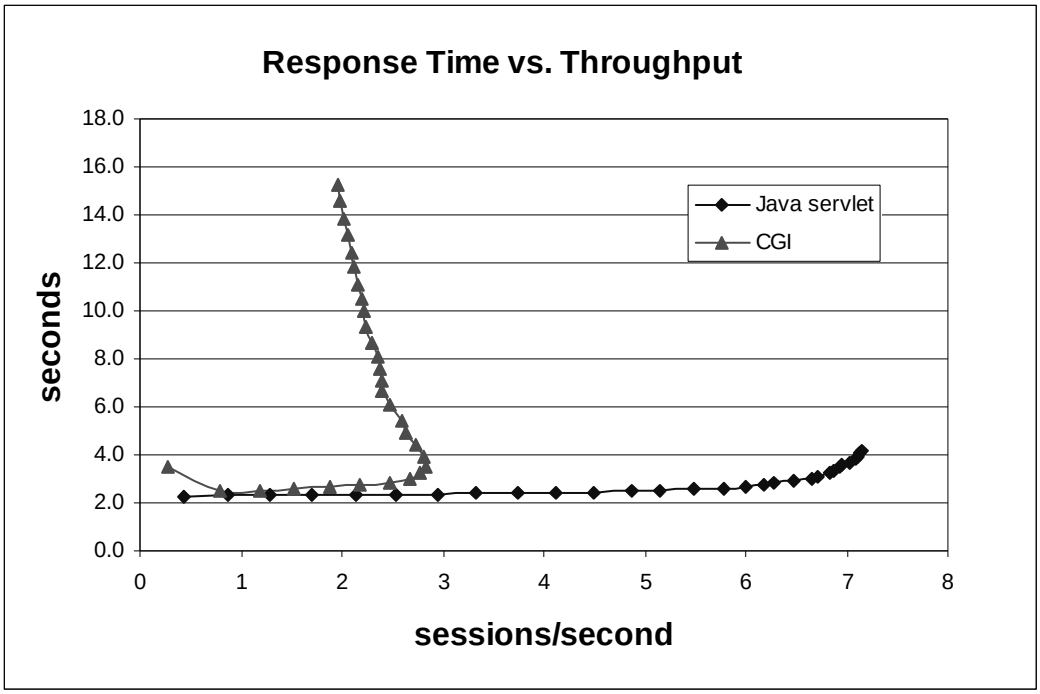

Figure 9: Response time vs. throughput comparison – trivial application

Technology	Maximum Throughput (sessions/sec)
Java Servlet	7.2
CGI	2.8

Table 5: Maximum throughput - trivial application

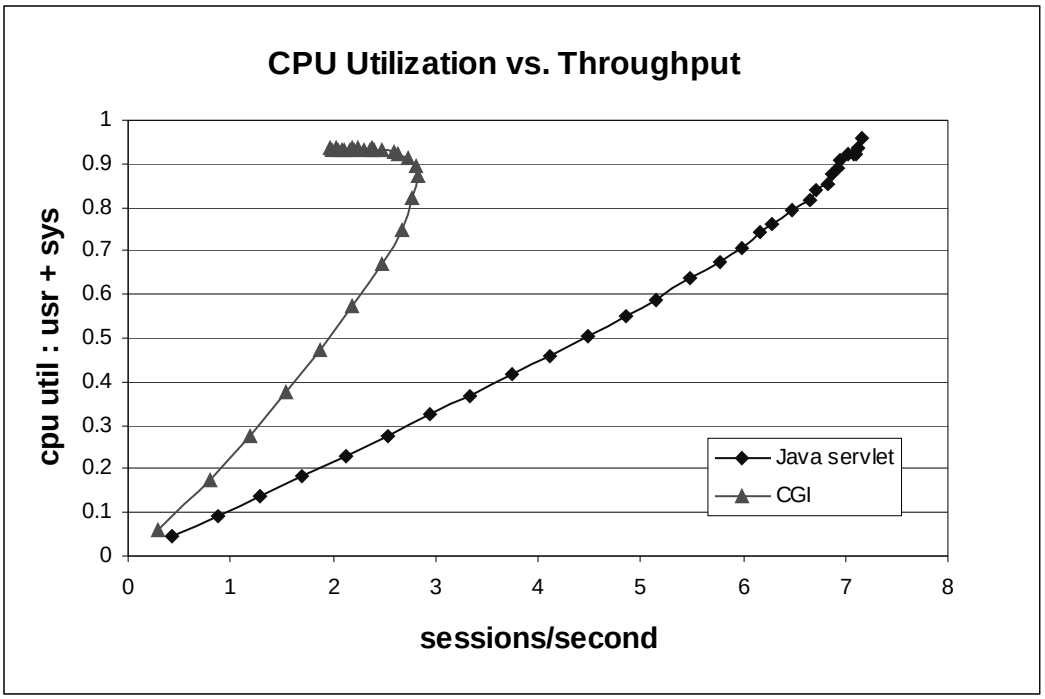

Figure 10: CPU utilization vs. throughput – trivial application

Technology	CPU time per transaction in ms	% of total CPU time spent in system mode	Actual ms in system mode	Actual ms in user mode
Servlet	237	38%	90	147
CGI	715	81%	579	136

Table 6: CPU milliseconds spent in user vs. system mode: trivial application

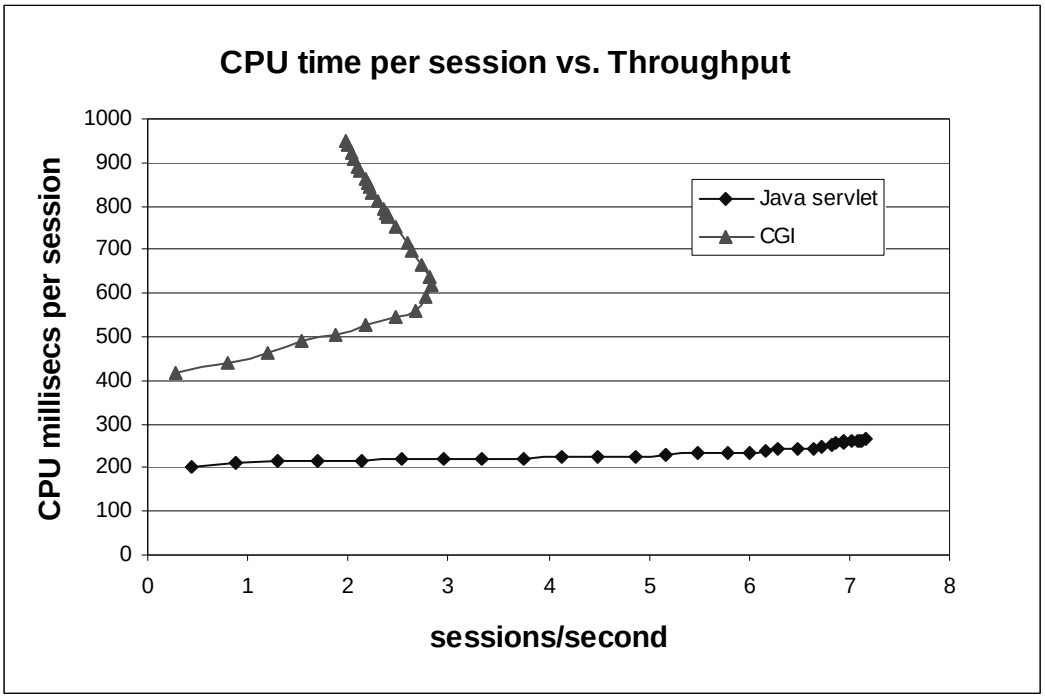

Figure 11: CPU ms per session – trivial application

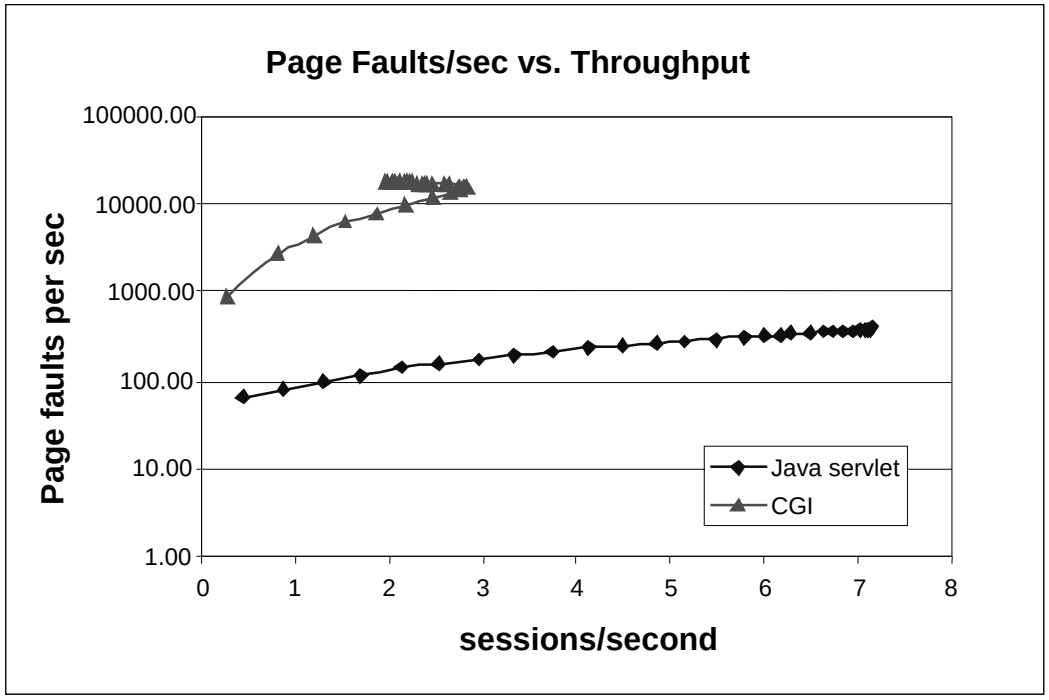

Figure 12: Paging activity comparison – trivial application

REFERENCES

1. T. Hansen, V. Mainkar, P. Reeser, “Performance Analysis of Dynamic Web Platforms”, in the Proceedings of the Symposium on Performance Evaluation of Computer and Telecommunications Systems, Orlando, Florida, July 2001.
2. “Java Servlet Technology”, http://java.sun.com/products/servlet/ .
3. B. Weiner, “iPlanet Web Server Enterprise Edition 4.0 and Stronghold 2.4.2 Performance Comparison”, http://www.mindcraft.com/whitepapers/iws/iwsee4-sh242.html, February 2000.
4. B. Kothari and M. Claypool, “Performance Analysis of Dynamic Web Page Generation Technologies”, in the Proceedings of the International Network Conference, Plymouth, UK, July 3-6, 2000.
5. A. Wu, H. Wang and D. Wilkins, “Performance Comparison of Alternative Solutions For Web-to-Database Applications”, in the Proceedings of the Southern Conference on Computing, University of Southern Mississippi, October 26-28, 2000.
6. P. Reeser, “Using Stress Test Results to Drive Performance Modeling: A Case Study in “Gray-Box” Vendor Analysis”, in the 17th International Teletraffic Congress (ITC), September 2001, Brazil.
7. “The Common Gateway Interface”, Internet document by the NCSA Software Development Group, http://www.w3.org/CGI.
8. “FastCGI: A High Performance Web Server Interface”, Technical White Paper by Open Market, Inc., http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm, April 1996.
9. “Java Server Pages”, http://java.sun.com/products/jsp.
10. A. Cockroft and R. Petit, “Sun Performance and Tuning”, 2nd ed., Sun Microsystems Press, 1998.
11. L. Kleinrock, “Queueing Systems, Vol. I: Theory”, Wiley-Interscience, 1975.