C Server Pages: An Architecture for Dynamic Web Content Generation

John Millaway
Dept. of Computer and Information Sciences
Temple University
1805 N. Broad St.
Philadelphia, PA 19122
millaway@acm.org

Phillip Conrad
Dept. of Computer and Information Sciences
Temple University
1805 N. Broad St.
Philadelphia, PA 19122
conrad@acm.org

ABSTRACT

This paper introduces C Server Pages (CSP), a highly efficient architecture for the creation and implementation of dynamic web pages. The CSP scripting language allows dynamic web contents to be expressed using a combination of C code and HTML. A novel feature of CSP is that the C portions of the CSP source file are compiled into dynamic load libraries that become part of the running web server, thus avoiding both the overhead of interpreted languages such as Perl, Java and PHP, as well as the need to create a separate process. We present an overview of the architecture and implementation, and provide results of performance benchmarks showing that CSP outperforms similar mainstream technologies, including PHP, mod_perl, and FastCGI.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems—World Wide Web*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/Server*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Design, Experimentation, Measurement, Performance, Languages

Keywords

CSP, C, scripting languages, web-based applications, performance, scalability, dynamic content, World Wide Web, C, CSP

1. INTRODUCTION

Many web applications require server-side generation of *dynamic content*, where the HTML presented to the user is the result of a computation that takes place on the server in direct response to the user request, as opposed to coming from a static disk file [13]. The Common Gateway Interface (CGI) was the first such technology [1]. Many others followed including JavaServer Pages (JSP) [2] (an extension of the Java Servlet concept), PHP Hypertext Processor (PHP)[6][1], mod_perl [5] and FastCGI [4, 15].

However, the currently available technologies have some drawbacks. The fork/exec model used in CGI scripts, for example, incurs heavyweight process creation overhead. Approaches that involve simple template substitution by the web server itself based on so called server-side includes (SSIs) are lightweight, but not very powerful. Application servers such as ColdFusion [3] introduce a heavyweight run time layer. Interpreted languages such as Java, PHP, and Perl are typically slower than compiled languages such as C. New languages specifically designed for the generation of server side content present an additional learning curve for already overburdened developers [8].

This paper introduces an alternative architecture called C Server Pages (CSP), based on combining languages familiar to many developers: HTML, and C. CSP pages consist of HTML combined with segments of C code to generate dynamic content. The web server invokes a compiler to convert the page into executable code in the form of a library that is loaded into a running web server and invoked when needed. CSP examines modifications times and status codes to minimize the number of compilations performed.

The CSP architecture offers several advantages. The first advantage is speed. CSP avoids both the overhead of creating a new process (fork/exec) to satisfy each request, as well as the overheads of interpreted languages such as Java, PHP, and Perl. The second advantage is expressive power; the full range of C features is available, as well as the full complement of system libraries that are available to C programmers. The third advantage is familiarity; CSP is based on HTML and C, technologies that are widely known; CSP adds only a handful of new tags for embedding the C code in the HTML. Therefore, the learning curve is reduced as compared to systems that require the programmer to learn a custom language.

This paper is organized as follows. Section 2 describes problems with existing web technologies. Section 3 describes the architecture of CSP. Section 4 describes related work. Section 5 presents results of benchmarks of CSP as compared with other technologies for dynamic web content. Finally, Section 6 provides a summary of our main results and suggestions for future work.

2. PROBLEMS WITH EXISTING TECHNOLOGIES

Early systems for dynamic generation of web content were built primarily with the Common Gateway Interface (CGI). The CGI is based on the fork/exec model in which the server must fork a child process for each request. Spawning a new process is a heavyweight operation in terms of response time and system resources. As a result, the fork/exec approach does not scale well. It has been shown that in-process or threaded servers respond significantly better than forked servers. [16, p. 728ff] Furthermore, with the fork/exec model, each new process is short-lived, and must ac-

[1]PHP is a recursive acronym standing for "PHP Hypertext Processor".

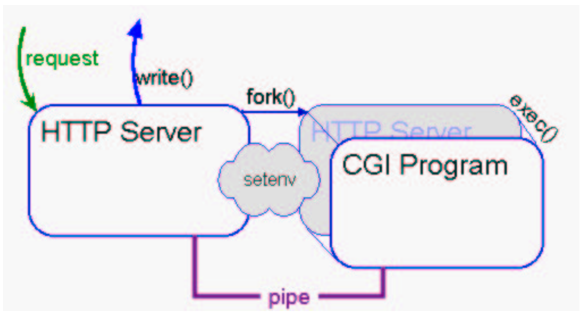

Figure 1: Fork/Exec Model for CGI

quire resources such as connections to a database upon each invocation, instead of caching those resources and sharing them between different request handlers. FastCGI addresses the overhead of process creation by maintaining a pool of CGI processes, and dispatching work to these processes via IPC, however this IPC itself is another source of overhead.

Many of the most popular and successful competing technologies (JSP, PHP, ASP, mod_perl, ColdFusion) are based on a *virtual machine* model. We include the persistent ”interpreted” language, mod_perl, in this model because its runtime interpreter caches first-time compilation results, very similar to a bytecode compiler targeting a virtual machine. In the virtual machine model, a request handler is compiled to bytecode, not to a native executable image. This bytecode is processed at runtime by a virtual machine. The bytecode processing is a runtime performance penalty that CSP aims to avoid.

Access to system libraries is often not straightforward in existing systems for dynamic content. Such libraries may be provided through complex bindings (for example, the Java Native Interface (JNI)), that introduce additional hurdles for developers. In the worst case, a library may need to be rewritten in the given language (if this is even feasible). In the case of PHP or ASP, the user is required to learn a rapidly changing and/or proprietary scripting language and run-time environment.

3. ARCHITECTURE OF CSP

3.1 CSP Syntax

The generation of the C source from the CSP source is a kind of inversion process, in the sense that the HTML source becomes embedded in the C source, instead of the other way around. Consider, for example, the CSP code shown in Figure 2, and a portion of the resulting C code shown in Figure 3. As can be seen, the segments of regular HTML code in the CSP source file are transformed into calls to `ap_rputs()`, a routine that performs an `sprintf()` like operation into the output stream for the HTTP response, while the C code, set off by the tags `<% %>`, is placed directly into the output file unmodified. The entire result becomes a single function call in the generated C code.

CSP provides six tags, as shown in Table 1. The first three insert C code into the output file. The `<% %>` tag inserts code directly at the point where the tag appears. The tag `<%* %>` relocates the enclosed code to a position outside the generated function call; this would be used, for example, for declarations such as `#include <somePackage.h>`. The tag `<%! %>` is similar, but inserts a variable declaration at the top of the generated function.

`<%  %>`	C code block
`<%*  %>`	global declarations
`<%!  %>`	local declarations
<%=*fmt, [args]* %>	`sprintf()` to HTTP output stream
`<%-- --%>`	CSP comment
<%@ *dir [attrs]* %>	CSP directive

Table 1: CSP Tags

```
<%@ page contentType=value
         httpHeader=value
         CC=value
         CPPFLAGS=value
         CFLAGS=value
         LDFLAGS=value
         LIBS=value
%>
<%@ include uri="uri" (static | dynamic) %>
```

Table 2: CSP Directives

The <%=*fmt, [args]* %> tag is used for output. The *fmt* and optional list of arguments are transformed into arguments to the equivalent of a `printf()` call that writes into the HTTP response stream. The `<%-- --%>` tag is used to mark comments at the level of CSP; it can be used to comment out sections of CSP code that span both C blocks and HTML blocks.

Finally, the `<%@ %>` tag is used for CSP directives. Two directives are defined, as shown in Table 2. The `page` directive provides two capabilities. First, it permits values to be directly output into the `Content-type` header or any other HTTP response header. Second, it allows control over various aspects of the C compilation. The `include` directive provides two ways for CSP files to refer to other CSP files. The static include provides a simple text insertion. The dynamic include is executed at runtime as a function call to a separate CSP shared object, compiled from the referenced URI. The effect of that function call is to insert text into the HTTP response stream at the position where the dynamic `include` directive appears in the source.

3.2 Request Cycle

Figure 4 shows the request cycle for CSP. The cycle starts with an incoming HTTP/1.1 and ends with the delivery either the requested resource or an appropriate error message. It is a “cycle” only in the logical sense that the request-response sequence repeats indefinitely. However, it is not a cycle in the true sense because, after one round of successful request-response operations, the CSP system does not return to the same state in which had been when it received the initial request. This state change in the request-response cycle is crucial to the mechanism by which CSP is able outperform existing technologies.

The request-response cycle can be logically divided into four distinct phases. The first phase, *URI Mapping*, determines what resource is being requested. The second phase is *compilation* in which the CSP source document is compiled to a native library. This is followed by the *link/load* phase to link and load the newly compiled executable into the running server. Lastly, in the *execute* phase, the library routine is invoked, producing either the requested resource, or an error. The compilation and link/load phases are activated only when the CSP source document has been modified more recently than the most recent compile, and link/load; otherwise they are skipped.

```
<html>
<head>
<title>CSP</title>
</head>
<body>
<h1>Sum of the Integers</h1>
<p> The sum of the integers from
1 to 10 is:
<%! int i,sum; %>
<%
   sum = 0;
   for (i=1; i<=10; i++)
     sum += i;
%>
<b> <%%d=sum %> </b> </p>
<p>Thank you for your attention.</p>
</body>
</html>
```

Figure 2: Example CSP file `sum.csp`

```
int i,sum;
ap_rputs("<html>\n"
         "<head>\n"
         "<title>CSP</title>\n"
         "</head>\n"
         "<body>\n"
         "<h1>Sum of the Integers</h1>\n"
         "<p> The sum of the integers from\n"
         "1 to 10 is:\n\n"
         ,r);

   sum = 0;
   for (i=1; i<=10; i++)
     sum += i;

ap_rputs("\n<b>",r);
ap_rprintf(r,"%d",sum );
ap_rputs(" </b> </p>\n"
        "<p>Thank you for your attention.</p>\n"
        "</body>\n</html>\n"
        ,r);
```

Figure 3: C code produced from sum.csp

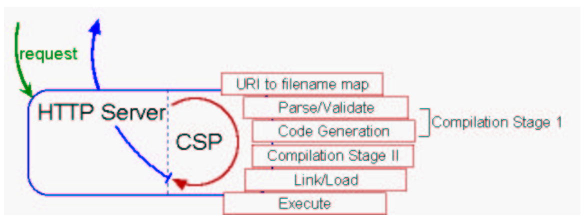

Figure 4: CSP Request Cycle

Each of these phases is described in more detail below.

3.2.1 *URI Mapping*

Incoming HTTP/1.1 requests refer to a particular resource, identified by a Universal Resource Identifier (URI) [7]. URIs that are sent in HTTP requests frequently refer to HTML source files on the web server, however, this need not be the case; in theory, a relative URI is abstract and does not necessarily specify an actual or physical resource, such as a file. Therefore, the purpose of the first phase in the CSP system is to identify the actual resource or service that is being requested, given an URI.

CSP maps a URI to both a CSP source file, and a resulting shared object containing an executable C function that, when invoked by the server, will produce the output needed. The CSP-enabled web server caches the results of compiling the CSP source, to avoid recompilation. Thus immediately following the URI mapping phase the CSP server must determine whether or not a compilation needs to take place to produce or update that shared object.

The following pseudocode outlines the decision as to whether a compile needs to take place for a given request. Assuming that `x.csp` is the name of the CSP file named by the URI, and `x.csp.so` is the name that would be given to the resulting shared object file.

```
if ( (x.csp.so exists) and
     (lastModification(x.csp.so) >
          lastModification(x.csp)) )
   return (noRecompileNeeded)
else
   return (recompileNeeded)
```

3.2.2 *Compilation*

Since the CSP language is a mixture of HTML and standard C, it requires a two-stage compilation process. The first stage invokes the CSP compiler, to convert CSP source into C. The second stage invokes the C compiler to produce an executable shared object (dynamic load library) in the format required by the server platform.

One detail here concerns what happens if the compilation fails; i. e., if there is a syntax error in either the CSP directives or the actual C code embedded in the CSP file. In this case, a stand-in shared object is produced that outputs an error message. Depending on whether the system is a development system or a production system, this error message might consist of either the actual error messages output by the compilation phase, or a message for the end user to contact technical support. (In the latter case, the compilation messages could be sent to a log file.) The important part is that a shared object file is produced with a modification time later than the source file; this suspends any further compilation of the offending CSP source file until it is modified by a programmer.

The second compilation stage is responsible for producing a dynamic link library, also known as a shared object, from the generated C code. For this, CSP is able to use any C compiler and linker combination that is capable of producing shared objects. In the current implementation, CSP delegates the creation of shared objects to the platform's GNU compiler suite, which produces ELF images (the Executable and Linking Format introduced in System V and widely implemented on various versions of Unix, including Linux.)

3.2.3 *Link/Load Phase*

The link/load phase is fairly trivial from an implementation standpoint, but we suggest from a conceptual standpoint, is one of the most interesting aspect of CSP.

The implementation is just two function calls: a call to the system function `dlopen()` to load the shared object produced by the compilation phase, and a call to the `dlsym()` function which returns a function pointer that can be directly called in the execution phase. The function pointer returned by `dlsym()` is placed into a hash table with the URI as the key.

Conceptually, this phase allows the runtime modification of a running web server. The resulting web server is able to directly provide new services without invoking any other process, subsystem, or virtual machine. The execution of the new capabilities takes place "as if" these capabilities had been a part of the original server source code, and compiled into the server from the beginning.

3.2.4 Execution Phase

During the execution phase, CSP must lookup retrieve the function pointer from a hash table, using the URI as the key, then invoke the function.

One detail here concerns what happens if the execution of the C code in the source file results in an exception (such as division by zero, segmentation violation, etc.) To handle this case, before invoking the function, the CSP enabled server defines signal handlers for each of the exception conditions that may occur. These signal handlers perform a long jump out of the offending code.

In spite of these precautions, it is still possible that the user generated code might cause an exception that cannot be handled. In the current Apache-based implementation, the Apache parent process manages several children, which handle the actual requests. A user-defined CSP routine will never be invoked by the parent; only by one of the children. The Apache parent process monitors the status of its children, and starts a new child process to replace any that die. If the rate of child death increases beyond a certain threshold, the parent can enable logging to determine whether a particular CSP module is the cause; if so, the parent can log this event, replace the shared object for that routine with a stub shared object that prints an error message such as "function temporarily disabled".

3.3 Security

The addition of CSP to a web server does not, per se, make it more vulnerable to external threats; of course, since the page author has the ability to program arbitrary C code, the code that is inserted could expose the server to new threats. One of the disadvantages of running code directly, rather than on a virtual machine or as a separate process, is that the code shares the same address space, and to a large extent, the same fate. This is however, not significantly different from the security and system integrity challenges facing the developers of any large C based application. The trade-off for additional power and speed is that the programmer must be quite careful, and stringent code reviews should be made of any CSP pages placed on a production server.

4. RELATED WORK

A detailed performance comparison of several technologies for dynamic web content is given by Gousios and Spinellis [9], which categorizes the available technologies by approach. Four main approaches are identified: CGI, Servlets, Templating, and Extension APIs. The CGI approach is identified as having poor scalability due to process-creation overhead, as lacking support for session-tracking in its design, and as suffering from the mesh of presentation logic and program logic. The Servlet approach is identified as extending the functionality of the containing application server, as having built-in support for session-tracking, and as having a distinct application server container, separate from the web server. FastCGI is similar to the Servlet approach in terms of the persistence of processes, and but different than the Servlet approach in the lack of a shared object space. The term "templating" is used to refer to several approaches that embed a scripting language or data representation language in HTML. Gousios and Spinellis cite code-maintenance issues as an impetus for the development of templating systems, and agree that templating systems are acceptable solutions for the development of content-based web sites and for web-publishing. Extension APIs are published C language interfaces to some exposed functionality of the web server. In this sense, one can write modules to extend the web server at compile-time. They cite the Apache module API as an example of an Extension API, and cite a well known problem with Apache's model: runtime persistence is not preserved across child processes in Apache's pre-forked server approach. They conclude that PHP does not scale well, and that FastCGI significantly outperforms the other approaches.

Iyengar et al. [10] shows that web server performance noticeably degrades as the proportion of dynamic pages to static pages is increased. The authors arrive at a similar conclusion as Gousios and Spinellis in [9], that is, because dynamic pages are costly as compared to static pages, it is crucial to optimize the processing of dynamic pages. Runtime models, such as Extension APIs or FastCGI, which improve the performance during requests for dynamic pages, should be chosen to replace traditional CGI models. Iyengar et al. [11] further defends the need for high-performance approaches to serving dynamic content and shows that a runtime process-caching approach yields the highest performance.

Kuhlins and Korthaus [12] defend servlets as a completely reworked solution to the undesirables of CGI. In particular, the authors cite security issues with CGIs that make CGIs impractical for use by Internet service providers. Servlets, they propose, provide a language-unified, elegant solution to certain problems, and if properly configured, outperform CGI. Kuhlins and Korthaus mention FastCGI but provide no performance comparison with servlets. They also mention briefly that the execution of a request handler in a Servlet environment is merely a method call, as compared to a process invocation in a CGI environment —but they do not expand upon this crucial point that is at the heart of CSP.

5. PERFORMANCE EVALUATION

5.1 Experimental Design

To evaluate the performance of CSP against competing technologies, we performed experiments to determine the maximum rate at which an Apache server (version 1.3.27) could satisfy requests using each of the following technologies:

- PHP 4.2.3
- mod_perl 1.0
- CGI
- FastCGI 2.2.2
- CSP 0.9

For each of the technologies under test, we created a script or module that would generate the same minimal HTML document (a title, one line of text, and the appropriate HTML markup). By using a minimal document, we focus this benchmark on the overhead of invoking each of the technologies, rather than on content generation. As a reference point, we also placed a copy of this document on the server and ran the same experiment to measure the performance of simple retrieval of static HTML.

We created a separate build of the Apache server for each of these technologies (except static HTML and CGI, for which we used the same build.) Although it is common to build a single server with support for multiple technologies, we chose this separation to eliminate the possibility of unpredictable interactions between technologies.

Our benchmark consists of a single server and multiple clients. By saturating the server with requests from the clients, we obtained the maximum number of responses per second sustainable by the server for each of the above technologies. Only one of the Apache builds was active for any given run.

The five Apache builds were tuned according to the recommendations of the Apache authors, and each technology was tuned according to the documentation provided. Specifically, we disabled the following Apache options: mod_dir, symlink checking, hostname lookups , AllowOverride , and ExtendedStatus. FastCGI was run with the `FastCGIServer` directive which causes the server to preallocate the CGI process pool; we configured FastCGI with 10 initial processes. We created a mod_perl handler from the mod_perl script in order to eliminate some processing overhead in the mod_perl internals. For PHP, we used the recommended default configuration. For the CGI benchmark, we chose to use a compiled C program rather than a Perl or CGI.pm based script, to provide the fastest possible time for CGI.

We used the 'httperf' tool as the load generator on the clients [14]. Httperf is designed specifically to measure web server performance, and is capable of generating repeated HTTP/1.1 pipelined requests. For each experiment, we ran httperf on k simultaneous clients with $k = 1 \ldots 5$. Each of the k clients made 1000 connections, with 100 pipelined HTTP/1.1 requests per connection. The key metric for the experiment is the maximum sustainable requests-per-second possible with each technology on the given server configuration.

The experiments were carried out using the emulab system provided by the University of Utah [17]. The client and server machines were all taken from a pool of 850Mhz Pentium IIIs, 512MB PC133 ECC SDRAM, Intel EtherExpress Pro 10/100Mbps Ethernet cards, and 40GB IBM 60GXP 7200RPM ATA/100 IDE hard drives. The emulab system was used to configure a VLAN consisting of the client and server machines. Throughout the duration of the experiment, there were no other users on these machines; they were devoted exclusively to the benchmark, and a separate test network was used to be sure that there was no interference from other traffic.

To ensure that the bottleneck was in the server processing rather than in the network, we examined the sizes of request and response packets. We noted that while the size of the generated HTML (i.e., the HTTP Content-length) was exactly the same for each technology, there were variations in the size of the HTTP request and response headers. This resulted in a 7% difference between the smallest and largest request messages, and a 17% difference between the smallest and largest response messages. However this relatively minor variation does not explain the large differences that were observed in performance among the technologies. We concluded that the network was not a bottleneck, because the bandwidth available on the VLAN exceeded the data rate necessary to support the fastest observed server response rate performance.

Inspection of both the Apache logs and the output from httperf verified that all requests were satisfied with a "`200 OK`" response.

5.2 Results

Figures 5 shows the average response rate of the server (in responses per second) for each technology. The x-axis is the number of clients, while the y-axis shows the response rate observed. Each data point shows the average server response rate (in responses per second) over the time required for each client to receive responses to 100,000 requests.

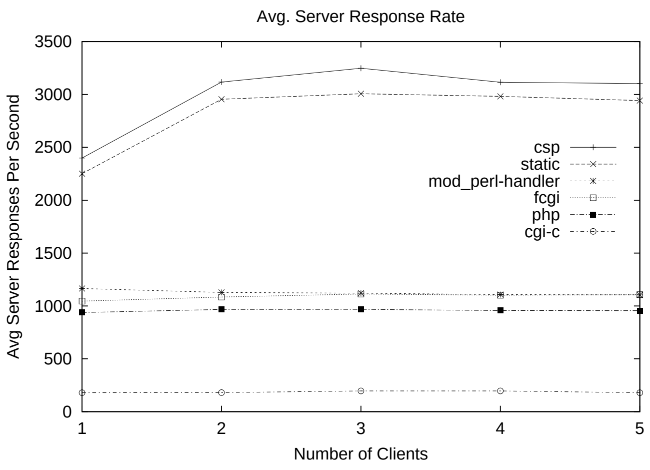

Figure 5: Benchmark Results

We conclude from the results that the load placed on the server by two or more clients was sufficient to reach the maximum sustainable server response rate. For all technologies except CSP and static html, a single client was able to sufficiently load the server. This is reasonable, given that the server and the client machines were of identical power, and that the client (httperf) has relatively little processing to do for each request. For CSP and static html, we can see that the results vary little from two to five clients.

CSP is the best performing technology, performing an order of magnitude better than CGI, and around three times faster than PHP, mod_perl and FastCGI. We suggest that the difference between FastCGI and CSP can be accounted for by the need to perform interprocess communication and a context switch between the web server and the FastCGI server process. PHP and mod_perl run in the same process as the server, as does CSP. The performance advantage of CSP over PHP and mod_perl is likely due to the speed of compiled C code vs. invocation of a bytecode interpreter.

Interestingly, CSP outperforms even static HTML, despite the fact that CSP is designed for dynamic content. This is due to the fact that CSP compiles the contents of a static page into the web server, so there is no actual transfer of data from disk before sending the response; the server need only do a `stat()` to check the modification time. When using CSP to serve static HTML, CSP effectively becomes a memory cache for the HTML. There is a drawback to using CSP to serve static content: there is currently no mechanism implemented in CSP to flush this cache if memory becomes scarce (for example, if a large number of CSP modules are loaded.) Solutions to this problem are discussed in Section 6.2.

6. CONCLUSION

6.1 Summary

CSP provides a solution for the generation of dynamic content that can significantly outperform existing approaches. CSP is based on adding a very small number of tags and directives to HTML and C, languages that are familiar to many developers. CSP also presents a new architecture for the generation of dynamic content, namely, a technique for directly extending the capabilities of a running server process with native compiled code.

6.2 Future Work

We plan to carry out additional performance evaluations of CSP in which the content is generated from a database, and in which CSP performs significant data manipulation on each request. We also hope to include JSP servlets in future benchmarks.

Planned enhancements to CSP include:

- support for C++
- the ability to store large chunks of static content on disk, rather than as string constants in the generated C code.
- support for periodic release of infrequently accessed shared objects via use of the `dlclose()` function.

7. ACKNOWLEDGMENTS

The authors are grateful to Jay Lepreau and the support staff of Netbed (formerly known as emulab), the Utah Network Emulation Testbed (which is primarily supported by NSF grant ANI-00-82493 and Cisco Systems) for making their facilities available for our experiments. We are also grateful to Andreas Jungmaier of the University of Essen for helpful comments.

8. REFERENCES

[1] Cgi.pm - a perl5 cgi library. http://stein.cshl.org/WWW/software/CGI/.

[2] Javaserver pages technology. http://java.sun.com/products/jsp/.

[3] Macromedia home page. http://www.macromedia.com.

[4] FastGGI home page. http://www.fastcgi.com.

[5] mod_perl. http://perl.apache.org/.

[6] Php hypertext processor home page. http://www.php.net.

[7] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax. Technical report, IETF Request For Comments, August 1998.

[8] Claus Brabrand, Anders Mller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology (TOIT)*, 2(2), May 2002.

[9] George Gousios and Diomidis Spinellis. A comparison of portable dynamic web content technologies for the apache web server. In *Proceedings of the 3rd International System Administration and Networking Conference SANE 2002*, pages 103–119, Maastricht, The Netherlands, May 2002.

[10] A. Iyengar, E. MacNair, and T. Nguyen. An analysis of web server performance. In *GLOBECOM '97*, 1997.

[11] Arun Iyengar, Ed MacNair, and Thao Nguyen. Web server performance under heavy loads. Technical Report RC 20976(92922), IBM Research Report, September 1997.

[12] Stefan Kuhlins and Axel Korthaus. Java Servlets versus CGI – implications for remote data analysis. In Reinhold Decker and Wolfgang Gaul, editors, *Classification and Information Processing at the Turn of the Millennium, Proceedings of the 23rd Annual Conference of the Gesellschaft für Klassifikation e.V.*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 841–847. Springer-Verlag, March 1999.

[13] Mike Morrison, Joline Morrison, and Anthony Keys. Integrating web sites and databases. *Communications of the ACM*, 45(9), September 2001.

[14] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[15] Peter Simons and Ralph Babel. FastCGI: The forgotten treasure. In *ApacheCon2001*, Santa Clara, CA, April 2001.

[16] W. Richard Stevens. *Unix Network Programming*, volume 1. Prentice-Hall, second edition, 1998.

[17] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (to appear)*, December 2002.