High-Performance Common Gateway Interface Invocation

Ganesh Venkitachalam   Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

{ganesh, chiueh}@cs.sunysb.edu

Abstract

As more and more Web services are delivered in the form of Common Gateway Interfaces (CGI) scripts, the efficiency at which Web servers execute CGI scripts is becoming ever more important. In this paper we show that the performance overhead associated with invoking a conventional CGI script could potentially become a bottleneck, especially for servers directly connected to high-speed network links. While existing CGI execution model runs CGI scripts as independent processes, the *LibCGI* architecture described in this work allows the Web server to execute CGI scripts as part of its address space. On a 100-Mbps Ethernet link and for web pages smaller than 10 KBytes, *LibCGI* is shown to be 2.3 times as fast as FastCGI, and 3.9 to 4.6 times faster than the conventional CGI model. This paper describes in detail the design and evaluation of the *LibCGI* architecture and its prototype implementation.

1 Introduction

Common Gateway Interface (CGI) is a standard service invocation mechanism that Web servers support either to provide "dynamic contents," HTML pages that are created dynamically to respond to user queries/requests, or to enact certain side effects in the background. CGI scripts[1], which are written in compiled languages like C or interpreted languages like Perl, add to the Web server's standard HTML page access service with a set of site-specific functions, ranging from database

[1] Performance-conscious Web servers typically use CGI scripts written in compiled language such as C, which is also the focus of this paper.

gateways to order processing in electronic commerce. In fact, the generic Web page access can be considered as a special type of CGIs that are built into the Web server by default.

As WWW/Internet permeates into the daily life of the mass, end users are demanding Web pages of higher qualities as well as more interactivity with Web contents. Statically composed HTML pages lack the flexibility to accommodate fast-paced content changes and the feedback capability essential to support on-line interaction. As a result, dynamically created HTML pages, which are results of program execution, emerge as the mechanism of choice in commercial Web sites. Therefore, the performance of invoking CGI programs should be at least as important as that of servicing URL pages when comparing Web servers. This is especially the case when the Web server is equipped with such a large amount of physical memory that client requests rarely require disk I/O, for example, Internet search engines.

Conventional CGI scripts are executed as separate processes that are independent of the Web server. Scripts are put in a special directory known to the Web server, and get invoked in response to URL requests requiring their services. Invoked scripts receive inputs from and return outputs to the Web server. The steps involved in executing a CGI script are as follows:

1. The Web server receives a request from a Web client.
2. The Web server `fork`s and `exec`'s the CGI script after setting up the environment variables according to the CGI Interface.
3. Once started up, the CGI script reads the environment variables and the standard input, which is usually tied to a pipe with the Web server, processes the request, and sends the output back to the server via the standard output, which is also usually tied to a pipe to the Web server. Then the CGI script process exits.
4. After parsing the headers to make sure that they are properly formed, the Web server sends the CGI output back to the requesting client.

The above CGI execution model is inherently slow because of the overheads associated with `fork/exec` and with inter-process communication for CGI scripts' inputs and outputs. This paper describes a new CGI execution model that allows CGI scripts to run in the same address space as the Web server. Consequently, the Web server invokes CGI scripts as if they are local procedure calls, thus completely eliminating the CGI invocation and inter-process communication overheads.

The rest of this paper is organized as follows. Section 2 reviews previous solutions to the CGI invocation overhead problem. Section 3 introduces the *LibCGI* architecture, which allows the Web server to call CGI scripts by making simple function calls. Section 4 describes the prototype implementation of *LibCGI* on the Apache Web server. Section 5 presents the results of a performance study on the *LibCGI* prototype. Section 6 concludes this paper with a brief outline of on-going work to further improve *LibCGI*.

	ProcessingLatency (μsec)
Fork	998
Exec	5250
IPC	932
Total	7180

Table 1: *Delay breakdown of the overhead of invoking a CGI script as measured on a Pentium 200-MHz machine.*

2 Existing Solutions

To have a better understanding of the conventional CGI execution model's performance overhead, we need to find out where the time goes when the Web server invokes a CGI script. The results measured on a Pentium-200MHz machine are shown in Table 1. All of these are (potentially) blocking system calls. Hence one should remember that the CPU might be doing useful work in the meantime. The Web server first forks a process, and passes the control to the forked process, which executes an `exec` call to start executing the designated CGI script. The input parameters are passed into the CGI process through environment variables. However, the outputs of a CGI script have to be communicated back to the Web server through an IPC mechanism. The Fork overhead was measured as the time it takes from the moment the parent process issues the `fork` system call to the moment when the parent gets back control. The Exec overhead was measured as the time it takes for complete the `exec` call. The Input/Output overhead was measured as the time from the moment a CGI process writes its output onto `stdout`, to the moment the Web server gets control, after being blocked on a read on that pipe. Note that the input overhead is included in the Fork overhead because the environment variables, through which CGI inputs are encapsulated, are copied from the Web server to the CGI process during the `fork` call. The last two rows in Table 1 also includes the CPU scheduler overhead. As we can see, the minimal CGI invocation latency turns out to be at least 7.18 msec . If there are multiple interactions between the Web server and CGI processes, the Input/Output overhead would increase proportionally.

2.1 FastCGI

FastCGI [3] is a CGI execution architecture that eliminates the `fork/exec` overhead by running CGI scripts as persistent processes that are constantly standing by to service client requests. FastCGI was originally developed to reduce the per-call start-up cost associated with CGI scripts that interact with relational database management systems. This type of CGI scripts typically require an initial hand-shake phase to establish states, which is usually rather time-consuming and involves IPC overhead. The steps involved in invoking the service of a FastCGI script are:

1. The Web server creates a FastCGI application process to handle a particular type of requests,

either at start-up or on demand.

2. The FastCGI program initializes itself properly and waits for the Web server to establish a socket connection.
3. When the Web server receives a client request of the target type, it opens a socket connection to the FastCGI process. The server then sends the CGI environment variable information and standard input over the connection.
4. The FastCGI process reads data from the socket, processes the request and sends results and/or error information back to the server over the same connection.
5. When the FastCGI process closes the connection, the Web server parses the returned headers, and sends the response back to the client.
6. The FastCGI process then waits for another connection from the server.

The Web Server and the FastCGI process communicate over Unix domain or TCP sockets. These processes use the FastCGI protocol to write and read the data from the socket. To hide the programming difference between generic CGI and FastCGI, a *FastCGI library* is provided to encapsulate all the interaction between the Web server and the FastCGI application process. Because there is no process fork and exec cost, the overhead of invoking a FastCGI script is reduced to 3.65 msec, and mainly comes from the control transfer between the Web server and the FastCGI process (1.9 msec), the inter-process communication for inputs/outputs (1.4 msec), and the FastCGI protocol processing overhead (0.35 msec). The FastCGI protocol processing itself requires scheduling back and forth between the Web server and the FastCGI application.

2.2 Web Server APIs

As Web servers become the generic front-end for offering WWW services, the software architecture of modern Web servers has been extended to support a flexible programming interface, which is designed to allow new extension services to be added in a modular fashion. Microsoft's ISAPI [4] and Netscape's NSAPI [1] and WAI [1], and Apache's module API [2] are examples of Web server API. Typical extension services include log collection, authorization, etc. The extension services are linked to the Web server as dynamic libraries and are running in the same address space as the Web server. Therefore, these APIs provide a good foundation to build an efficient CGI invocation mechanism. Unfortunately, the current design of these APIs do not directly support simple CGI Invocation. These web server extensions are meant to be used for more complicated purposes than CGI execution. Hence the complexity of programming CGI scripts in such a fashion is significantly more than that of programming conventional CGI scripts. Also, the name space for Web server extensions is not exposed to Web clients, as in the case of CGI, and the APIs are much more complicated compared to CGI programming model.

With `mod_perl` [7] it is possible to write web server extensions using Perl. These extensions can hook into the Apache server, or just work as CGI scripts. The idea is to embed a Perl interpreter into the Apache Server, so that the cost of forking and exec-ing a Perl interpreter each time to service a CGI request is avoided. In this respect, the idea is similar to FastCGI. But the fact that interpreted languages like Perl are slower than compiled languages still remains, especially since Perl is directly interpreted rather than from an intermediate byte-code form.

WAI [5]is a distributed CGI system based on CORBA and it is easier to use than the NSAPI and ISAPI. The basic idea is to have a WAI service running, which accepts requests and sends back results to the Web Server. In the case of WAI, the complexity of communication between the Web server and the WAI service is hidden behind the CORBA interface. But the generality of WAI again introduces the IPC and data copying overhead, as in the case of FastCGI.

The proposed LibCGI mechanism is designed to support a programming interface similar to CGI, and is built on top of Apache's module API, which is comparable to ISAPI and NSAPI in terms of the level of programming abstraction.

3 Faster CGI Invocation using LibCGI

3.1 Architecture

As we have seen, FastCGI retains the ease of programming associated with CGI, while cutting away the Fork/Exec overheads. Web Server APIs do away with even the IPC overhead, but are too complicated for normal use as the base for CGI program development. Ideally, what we want is to preserve the CGI programming model, while keeping the CGI invocation cost comparable to local function calls. The basic idea is to compile CGI scripts into a shared library and load the shared library upon a client request. Once a shared library is loaded a specific function which is exported by the library will be called to process every request. The loading cost itself is a one-time cost because the library is unloaded only when the Web server exits. Thus when a request for the service of a CGI script that is already loaded arrives, the CGI invocation overhead is the same as that of making a local procedure call, which does not involve any blocking system calls.

We assume that CGI script is written in a compiled language and compiled into a shared object, say, (`.so`). LibCGI can load these CGI shared libraries dynamically, and process a CGI request by calling the corresponding shared library. The key advantage of LibCGI is that its programming model is very similar to the conventional CGI programming model. In particular, there is no need to recompile the Web server each time a new CGI script is installed under LibCGI. And the API for writing CGI scripts, including the name space visible to Web clients, remains largely unchanged.

3.2 The LibCGI programming model

The source code of a LibCGI script is compiled as a shared library. Conventional CGI scripts are put in a `cgi-bin` directory, but LibCGI shared libraries are placed in a separate directory,

e.g., `libcgi-bin`. Installation of a LibCGI script is nothing but copying the shared library to this directory. Because LibCGI scripts are running in the same address space as the Web server, only trusted personnel have the authority to install a LibCGI script can be installed. The name of the CGI shared library is specified as part of the URL. For example, to invoke a LibCGI script that has been compiled to a shared library named `run.so` and put in a directory `libcgi-bin` under the document root of the Web server, *www.cs.sunysb.edu*, clients should send a URL request that looks like `http://www.cs.sunysb.edu/libcgi-bin/run.so`. When the Web server receives such a request, it sets up the necessary environment variables and makes a call into the corresponding dynamic library to perform the requested function, with an input parameter containing the string received from the client. This string would be available to an ordinary CGI script from `stdin`. The called library can access the environment variables using the standard `getenv` function. The library performs the requested processing and writes the resulting output back to the client using a `lwrite` function that we provide, instead of the standard `write` function. The `lwrite` function is the interface between Apache and the CGI script in the LibCGI model. `lwrite` processes the output produced by the CGI library.

Programming CGI scripts in the LibCGI model is exactly the same as programming a conventional CGI script except the following changes:

- An *initialization* function `initfunc` whenever appropriate, should be provided, which is called by the Web server when the LibCGI script is loaded the first time around. Such initialization steps as establishing a database connection or reading g configuration files should be done here.

- Every LibCGI script has to export a `processfunc` function. This function will be invoked on every request that the Web server gets for his script. The `processfunc` is thus essentially the equivalent of `main()` in a conventional CGI script written in C.

- LibCGI scripts do not read from the `stdin`. Instead the Web server relays the input from the client as a string input argument when it calls a LibCGI script as a function.

- LibCGI scripts use the `lwrite/lputs` functions instead of `write/puts` functions as in the conventional CGI model.

- When a LibCGI script is unloaded, the Web server will call a `terminatefunc` function `terminatefunc` if available, to perform any necessary cleanup.

Only `processfunc` is required in every CGI script as it is the function which is called to do the required processing on every incoming request. However, `initfunc` and `terminatefunc` are optional.

4 Prototype Implementation

4.1 Basic Software Architecture

Apache is designed to be extensible through the concept of `modules`. A new module can be linked into the Apache Server to implement whatever additional services desired, thus extending the Web server's functionality An Apache module's interface to the core part of Apache is via the `module` structure. This structure exports a set of function pointers to the Apache core, which will be called at different times during the processing of a request. For example, a module can provide functions to be called for parsing the HTTP Request header or for servicing the actual request. One can add a handler for a new type of HTTP objects by exporting a handler function via the `module struct` and adding that function and type in the configuration files. A requested file's type is determined by the filename extension that the Web server gets as part of the URL. The Apache core searches through the configuration files to find out the appropriate handler for each object type, and invokes the appropriate handler routine from the table of function pointers exported by the corresponding module. Also, new modules can use Apache-provided functions like `send_header, lwrite` and `lprintf` to send back the parsed header and data results back to the client, and so on.

The basic idea of our implementation strategy is to create a new Apache module, called the *LibCGI module*, specifically for handling HTTP requests of the type `.so`. When a request of the type `.so` is received by the Apache server, a new handler implemented in the Apache core will be invoked to handle this request. The parameters of the client request are passed to this handler via the `request_rec` structure. The handler will load the LibCGI module as a shared library if it is not already loaded. Apache keeps track of loaded libraries in a list. The processing function `processfunc` of the LibCGI shared library is then called after Apache saves the current environment, sets up the environment variables for the shared library and reads the client input data if any.

The function `processfunc` is passed a single pointer that points to the data received from the client as a result of a POST Method, or a NULL pointer if the requested operation was a GET method. Then `processfunc` proceeds to process the client request in the normal fashion, and finally uses `lwrite/lputs` functions to send back the output. The actual details of sending HTTP response headers etc., is kept transparent to CGI script programmers because the headers returned by the LibCGI library function are parsed and stored in the standard data structures of the Apache server. If there is an error in the header, further outputs from the script is ignored, an *Internal Server Error* message is sent back to the client, and the corresponding LibCGI library is unloaded.

We have implemented a *LibCGI* prototype on Apache 1.2.5, on a Pentium 200 machine with 32 MB RAM, running Linux 2.0.34 We used the `dlopen, dlsym and dlclose` functions provided in Linux and GCC version 2.7.2.3. As most UNIX platforms have a similar mechanism to support the creation and loading of shared libraries, porting the implementation prototype to other UNIX platforms is expected to be relatively straightforward.

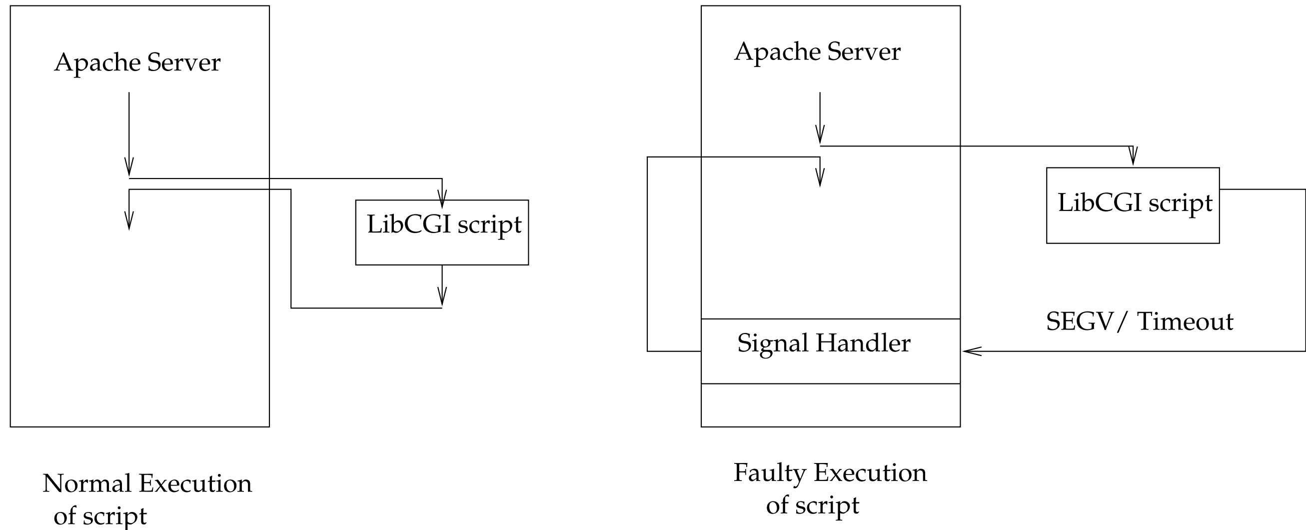

Figure 1: *Comparison between LibCGI execution flows in the presence and absence of errors.*

4.2 Error Processing

If everything proceeds normally and the script execution was successful, the output of the function is received by the client, and the library will persist across requests. After the first time the script is accessed, the cost of executing the script is the cost to set up environment variables and the cost of a function call. Data copying between the Web server and the script is also reduced, because they run in the same address space. Thus the LibCGI invocation overhead is minimal and roughly equivalent to to that of a local procedure call, which is about 20 CPU cycles, or 0.1 μsec on a 200-MHz machine. overhead is minimal.

If a LibCGI script were to go into an infinite loop, this would actually prevent the Web server itself from running, and thus denying the service to all subsequent requests. We use a timeout mechanism to detect long-running LibCGI scripts, and terminate them if necessary. Similarly a segmentation violation by a LibCGI script would actually terminate the Web server. This is handled by installing a signal handler for the segment error signal, which would unload the errant LibCGI library and return control to the Web server. Here care must be taken to write the termination function in the library so that it does not assume anything about the state of the library. For example, the termination function may be called because of some error caused by the processing function, `processfunc`. Figure 1 compares the execution flow of a LibCGI script in the faulty and fault-free cases.

If the processing function times out, or causes a segmentation violation, an *Internal Server Error* message is also sent back to the client. However, because part of the headers/output of the LibCGI script might have been already sent to the client, any such errors should be logged in the Server error logs for subsequent analysis.

One major drawback of the LibCGI mechanism is that interpreted scripts like Perl or shell

scripts cannot benefit from this approach to improve their performance. Only programs which can be actually compiled into a shared library can take advantage of LibCGI. Fortunately, CGI applications that are conscious about performance are almost always implemented in a compiled language. There is another issue regarding the performance of CGI script execution. If the Web server chooses to set up the `REMOTE_HOST` environment variable for the library, it involves a reverse DNS lookup, which could potentially cost seconds and wipe out all the performance gain of LibCGI. This is not specific to LibCGI, but a well known problem in general. Hence in all the following measurements, the Apache Server was configured such that this environment variable is *not* set up. Nevertheless, we export a function `get_remote_host` to LibCGI scripts, which returns the host name of the source node by performing a reverse DNS lookup. As a result, only those CGI scripts that need this information need to pay the DNS lookup overhead. As mentioned above, if the LibCGI library code causes a segmentation violation or timeout after sending correct headers, partial output might have been send back to the client. In such a case, the Web server sends an error message to the client as an explanation for the anomaly.

4.3 Protection and Availability

Because the Web server and the CGI scripts reside in the same address space, the Web server is not protected from buggy CGI scripts. The LibCGI architecture incorporates a three-pronged solution to address this problem:

1. An intra-address space protection mechanism based on segmentation hardware is used to establish the protection boundaries between CGI scripts and the Web server.
2. The Web server is modified to catch segmentation fault and timer expiration signals protect itself from CGI scripts that cause segmentation violation errors or enter infinite loops.
3. A separate process that constantly monitors the health of running Web servers, and restarts a new Web server when one of time crashes, which is already built into the Apache Server. In addition, only trusted CGI scripts are allowed to run in the LibCGI mode.

The second solution provides a minimal defense against typical programming bugs in CGI scripts. The third solution is relatively straightforward, as the Apache server follows a process-pool model. That is, several Web server processes are pre-forked and are able to servicing requests concurrently. Hence when one of the server crashes, there will always be other Web servers available for handling the next request for a given LibCGI script. Because these Web servers processes reside in separate address spaces, the crashing of a server process due to a buggy LibCGI script does not affect the availability of the Web service as a whole.

Although the last two solution are sufficient for all practical purposes, they are not completely satisfactory as they only address LibCGI's availability problem, leaving the protection issue unresolved. We have developed a novel intra-address-space protection mechanism using Intel's x86 segmentation hardware [6], which is briefly described in the following paragraphs and whose details

are beyond the scope of this paper. The current LibCGI prototype, however, does not include this mechanism yet.

We use a combination of segmentation and paging protection hardware in x86 architecture to support safe user-level extensions. In this case, the Web server is the application and the CGI script is an extension to the application. The basic idea is to load the application and the extension into different segments of the *same* virtual address space (VAS), each with a different privilege level. The Intel processor provides protection check at both segment and paging levels. Different Segment Privilege Levels (SPL) are mapped to different Page Privilege Levels (PPL). An application process starts at a particular SPL (say Level 3) and converts itself to a higher SPL (say Level 2) through a system call. All the pages owned by this application are also now marked at a higher PPL. Protection between the OS kernel and the application still exists because the kernel and the application have different SPL.

When the first user extension is loaded, an extension segment residing in the same VAS as the application is created, but with a lower SPL and PPL than the application. Pages not currently used by the main application are allocated to hold the user-level extension's code, stack, and data. In addition, the application can choose to expose to the extension whatever pages it desires. This means that the application can share selective portions of its VAS, such as shared libraries or shared memory regions. As a result, a user-level extension can access its own code, data, and stack, as well as libraries and memory regions in the main application. However, user-level extensions can never access pages that the main application chooses to hide. Because the main application process' segments cover the same virtual address space range as the extension segment, relocation supported by a generic dynamic linker such as `dlsym` is sufficient. Moreover, data/function pointers can be passed between the main application and extensions without modifications, thus greatly facilitating code/data sharing and reducing unnecessary memory copy (`memcpy`) overhead. Most of all, this protection mechanism is almost transparent to the application programs. The only modifications necessary to the application are the addition of a system call in the beginning to change its protection level, and replacing `dlopen` and `dlsym` calls with `seg_dlopen` and `seg_dlsym`, which are LibCGI versions of `dlopen` and `dlsym`. Optionally, applications need to make additional system calls to make available those memory regions chosen to be exposed to the extensions, with pointers to the memory regions as parameters.

5 Performance Evaluation

To compare the performance of CGI, FastCGI, and LibCGI, we measured the latency of invoking a CGI script under these three architectures. We instrumented the Apache server to measure the time it takes to handle a complete request for each CGI execution model. We created a CGI, FastCGI and LibCGI program that does nothing but to write a memory-resident data packet of various size back to the requesting client. Then we measured the time between when the Web server receives a client request and when the data packet is completely sent out. The resulting

	Request Handling Time		
Size of Data sent to client	**Conventional CGI (msec)**	**FastCGI (msec)**	**LibCGI** (msec)
28 Bytes	8.2	5.0	1.3
1 KBytes	8.6	5.3	1.6
Hl *10 KBytes*	11.6	7.5	2.4
100 KBytes	75	71	63

Table 2: *Comparison of the execution latency of a program that returns a memory-resident data packet of various size to the requesting client under the conventional CGI, FastCGI and LibCGI architectures.*

	Throughput (requests/sec)							
Size of HTML file requested	**Conventional CGI**		**FastCGI**		**LibCGI**		**Web Server**	
Bit Rate (Mbps)	10	100	10	100	10	100	10	100
28 Bytes	88	98	162	193	320	448	323	460
1 KBytes	81	92	151	188	278	431	285	436
10 KBytes	52	76	72	130	85	312	86	315
100 KBytes	9	38	10	47	10	88	10	89

Table 3: *Comparison of CGI, FastCGI and LibCGI in their execution throughput as measured by numbers of scripts completed per second, assuming that the Web server and its clients are connected through a 10-Mbps and a 100-Mbps Ethernet link. All HTML files accessed during the tests are memory-resident.*

latency measurements are shown in Table 2. All the performance measurements were collected on an Apache Web server running on a Pentium 200-MHz machine with 32 MB RAM.

LibCGI is about 3.8 and 6.3 times as fast as FastCGI and CGI respectively, when the data packet size is 28 bytes. The speed-up ratio decreases as the size of data packets increases, because the data transfer time accounts for a progressively more significant portion of the total latency. However, even for 10-KByte packet, the corresponding speed-up factors are still rather significant, 3.1 and 4.8, respectively. Compared to CGI and FastCGI, LibCGI costs an additional one-time overhead of loading a CGI script as a shared library, which is about 400 μsec.

While LibCGI delivers impressive improvements over FastCGI and CGI in CGI script processing latency, this does not necessarily translate to end-to-end server throughput improvement if the network is the bottleneck. We used the ApacheBench[2] benchmark included in the standard

Apache distribution to measure the overall throughput of different CGI execution models, when the Web server and its clients are connected through a 10-Mbps and a 100-Mbps Ethernet link. In each run, a total of 1000 requests are sent to the Web server with up to 30 requests being serviced concurrently. Each request involves an access to a fixed HTML file, which is memory-resident at all time. The Web server can service each request directly by opening the HTML file, reading it in, and writing it back to the requesting client. The Web server can also service each request by invoke a CGI script that does exactly the same thing but under different CGI execution models. Table 3 show the throughput statistics when the requests are serviced by the Web Server directly and by CGI scripts under various execution models. The Ethernet links are quiescent in all runs.

The Web Server column in Table 3 establishes an upper bound on the CGI script execution throughput when network delay is included, as there is no CGI script invocation overhead in this case. For all data sizes and network speeds, LibCGI is within 3% of the corresponding configuration's bound, conclusively demonstrating its effectiveness in reducing the performance overhead of invoking CGI scripts. On a 100-Mbps link and when the HTML file is smaller than 10KBytes, LibCGI is about 2.3 times faster than FastCGI, which in turn is 1.7 to 2 times faster than CGI. These throughput improvements are significant because most popular Web servers are connected to the Internet through links with an aggregate bandwidth higher than 100 Mbps, and most Web pages are rarely larger than 10 KBytes. The gaps between LibCGI and FastCGI, and between FastCGI and CGI, decrease as the HTML file increases and/or as the link speed decreases, because the non-invocation overhead becomes more significant when the file is larger and/or when the network link is slower.

6 Conclusion

This paper describes the design, implementation, and evaluation of a high-performance CGI architecture called *LibCGI*. The main idea behind the LibCGI architecture is the use of the dynamic library linking and loading capability that Apache's module interface provides. By loading CGI scripts into the same address space of the Web server, the overhead of invoking a LibCGI script is reduced to that of a local procedure call. Performance measurements on a working prototype show that LibCGI improves the the CGI script execution throughput over FastCGI by a factor of 2.3, and over the conventional CGI model by a factor of 3.9 to 4.6. The LibCGI prototype we implemented also provides a primitive set of protection mechanisms that keeps the Web server from being affected by such bugs in LibCGI scripts as infinite loops and segmentation violation.

Currently, we are working on an intra-address space protection mechanism that exploits segmentation hardware available in Intel's x86 architecture to establish protection boundaries among software modules within the same address space [6]. This mechanism, once fully working and incorporated into Apache, will provide an efficient and effective way to support the LibCGI architecture without safety concerns. We are also exploring the idea of applying the LibCGI architecture to other types of service extensions, e.g., CORBA object invocations.

References

[1] NSAPI Programmers Guide, http://developer.netscape.com/docs/manuals/enterprise/nsapi/

[2] The Apache Server project, http://www.apache.org/

[3] The Unofficial FastCGI Homepage, http://fastcgi.idle.com/

[4] ISAPI Reference, http://www.microsoft.com/win32dev/apiext/isapiref.htm

[5] Writing Web Applications with WAI, http://developer.netscape.com/docs/manuals/enterprise/wai/

[6] Chiueh, T.; Venkitachalam, G.; Pradhan, P.; "Intra-Address Space Protection using Segmentation Hardware", Seventh IEEE Workshop on Hot Topics In Operating Systems (HoTOS - VII), March 1999, Rio Rico, Arizona.

[7] Apache/Perl Integration Project, http://perl.apache.org/