2 Solutions for CGI Optimization

Inspite of the shortcomings mentioned in the above section, all is not lost. In the next sections, we will discuss the remedies and point out various techniques that can improve CGI performance in various cases.

2.1 Efficiency in Perl

The first step towards CGI optimization is obviously to follow the guidelines outlined for efficiency in the language of choice itself (which in our case is Perl). For frequently accessed CGI's, using a Perl compiler (distributed with Perl 5.005) that generates C code from Perl scripts, can significantly improve efficiency as there is no overhead of starting up the Perl interpreter. Perl performance can also be significantly improved (and in certain cases, even better than the compiled C code) when using mod_perl module in Apache.

CGI scripts, such as the ones using *system()* (which also reduces portability) or ***`backtick`*** notation are inefficient by their very nature, and very resource-intensive. There are ways to reduce or eliminate all these overheads, but these tend to be operating system- or server-specific (for which the best support seems to be in Apache).

2.2 I/O Buffering

I/O buffering has its advantages but for time-intensive computations, (for example, searching a large database or creating images on-the-fly), it can be a bottleneck. You could follow these steps to adjust I/O buffering:

1. Turn off I/O buffering in Perl by using *$|* = *1*.
2. Send the header information (at least the ***content-type***) to the browser (else the browser will go into a timeout and close the connection).
3. Turn on I/O buffering in Perl by using *$|* = *0*.
4. Send the content.

2.3 Reverse DNS Lookups

The server is given only the IP address of the browser making the request. The reverse DNS lookups let the server use the full qualified name in CGIs. The problem with DNS is that it uses blocking systems calls which hang the (parent) server process till a call is completed. These calls can take a significant amount of time for a single user, resulting in a sacrifice in performance, if many users are being served.

Explicit reverse DNS lookups are not needed as, if needed, CGIs can do a lookup themselves using the environment variable. If possible, avoid runtime reverse DNS lookups and use static IP addresses.

In some servers, such as recent versions of Netscape Enterprise, DNS lookups are set off by default. To turn off reverse DNS lookups in Apache, you can do the following in *httpd.conf*:

```
 HostnameLookups = off
```

and the following in the *AddLog* directive:

```
 iponly = 1
```

2.4 Non-Parsed Headers

Most Web servers buffer the output from the CGI script before sending it onward to the browser. If the buffer size is large and the size of a page is small, then the script may have to send several pages before the first one is sent to the browser, resulting in choppy updating.

When a *Content-type* header is included in a CGI script, the server *parses* the output and *completes* the header information (by adding the header information of its own) that it considers may be useful to the browser. However, CGI scripts can override the header information included by the server by generating a complete HTTP header on its own. CGI scripts which bypass the server and generate the HTTP header information on their own are known as non-parsed header (NPH) scripts.

The advantages of NPH CGI scripts are:

- In contrast to ordinary CGI scripts, they can keep the connection between the server and the browser open, and can output results over a relatively longer period of time.
- They are slightly faster in response time for the user than ordinary CGI scripts, since when a script returns a complete HTTP header, the output is presented ***directly*** to the user. There is no interference on part of the server and hence no overhead.

The limitations of NPH CGI scripts are:

- Including correct header information is the script developer's responsibility. If there are any errors in the header information, the server will not be able to circumvent them and the browser will not be able to interpret the output.
- The server can not log the size of the data returned through an NPH CGI script.

NPH scripts can be useful in instances that require "server-push." Examples are animation programs that need to induce "continuity" when presenting image frames to the user, and stock pricing programs which depend on constantly changing data.

2.5 Division of Labour

For multiple CGI's, an improvement in scalability can be achieved by running them on different Web servers. If the CGI's do share data, then just the CGI's can be placed on different systems. If the CGI is being used as frontend to other applications, such as a database, then the backend program should be run on a separate server doing most of the work, while the actual CGI simply carries messages.

2.6 Client-Side Processing

CGI is a server-side technology. For a task at hand, such as form validation, a CGI doesn't have to do all the work involved in the process. The work could be shared with client-side technologies such as JavaScript. Moving some of the processing from the server-side to the client-side by supplementing CGIs with client-side technologies has various benefits:

- The browser, which spends most of its time idle, waiting for incoming requests. By moving some of the work onto the browser reduces the amount of work servers do, and hence the load on the server.

- If used for validation, it eliminates wasted CGI calls due to invalid input from the user. Such user-centric validation can save enormous time. The CGI itself can then be smaller. Besides validation, JavaScript could also be used for light calculations on the client-side.

This solution also has certain limitations:

- Not all browsers support JavaScript.

- The trade-off (though small) is that the user has to download a little more data.

- There are different JavaScript implementations in different browsers. Incompatible implementations can not only be inconvenient, but even lead to the possibility that a JavaScript script may not work at all. One way to circumvent this is for the CGI to generate all the JavaScript code in the application, and use the *USER_AGENT* environment variable to serve customized JavaScript. If the CGI script detects that the user's browser does not support JavaScript, it can generate Web pages that do not require JavaScript at all. (This, however, does not reduce the size of the CGI script, which, as mentioned above, is a benefit of client-side processing.)

- The users can turn-off JavaScript-support in their browsers at *any* time.

  JavaScript stops functioning if the user runs out of memory, which can mislead the server to conclude that the input has been validated. In this case, the CGI script need to check if the JavaScript is running, and take appropriate action accordingly. One way to do that is to have JavaScript post the form to the CGI. So if JavaScript is not running, the form cannot be posted. It includes a hidden form field which tells the server whether client-side validation has been performed.

Thus, JavaScript can not entirely remove the burden of validation from the server but, in certain cases, it can reduce it.
Note that, one could also use VBScript for client-side validation but it is only supported on Internet Explorer; JavaScript is supported on many more browsers and is moving towards a standardization as ECMAScript (as defined by ECMA-262).

2.7 State Persistence Using Cookies

Cookies can eliminate repetitious validation of user information or their state, so that the CGI does not have to look it up each time a page is accessed. The limitation of using cookies are that the user can refuse cookies (for example, for reasons of privacy), they are limited to 4K, and HTTP 1.0-based browsers do not support them.

2.8 Co-Processing

One way to avoid latency of CGI scripts, is to keep them running all the time as a co-process. Intead of having CGI start in response to a query and die, it can be useful to start-up a persistent CGI-like process along with the Web server. When the Web server gets a request pointing at that process, it connects to the process, hands over the request, and waits for the response while still being able to handle other requests.
One obvious limitation of co-processing is the risk of memory leak since the process has to run all the time. Chances of this can be reduced with utilities that can detect and locate the problematic areas in the script.

2.8.1 FastCGI

An alternative to the CGI protocol is the FastCGI, a standard protocol proposed by Open Market, Inc. The idea behind FastCGI is co-processing. FastCGI is a simple communications protocol that works as follows: it uses a single TCP socket to connect the Web server and the FastCGI script (in contrast to the ordinary CGI method of using pipes and environment variables). This connection provides a CGI-like environment and other (I/O streams, error-specific) information, which is set-up at the beginning of each request. (The environment variables and ***stdin*** data is directed to the application, and ***stdout*** and ***stderr*** data is directed to the Web server.)

The Web server runs FastCGI scripts as separate processes like ordinary CGI scripts. However, once launched, these scripts don't immediately exit when they finish processing the initial request. Instead, they go into an infinite loop that waits for new incoming requests, processes them, and goes back to waiting.

The advantages of FastCGI are:

- FastCGI avoids CGI's problem of having to launch a new script to handle each and every incoming request by keeping the connection open at all times. It creates a single persistent process for each FastCGI script which eliminates the need to create a new process for each request. So, unlike CGI, you do not have the overhead of starting up a new process and doing application initialization (such as, connecting to a database) each time somebody requests a document. By keeping application processes running between requests, FastCGI gets a better performance than CGI. Here is a FastCGI benchmark demo.

- FastCGI programs are scalable since they can run off systems different than the Web server. An application can reside on a different machine from the Web server, allowing applications to scale beyond a single box and providing easier integration with existing systems.

- Like CGI, FastCGI applications can be written in a variety of languages, including Perl, C, C++, Java, and Python.

- Existing CGI scripts can be adapted to use FastCGI by making a few changes to the script source code. (For example, the Perl 5 CGI library, CGI.pm, provides a simple way of doing that.)

Implementations of FastCGI in Apache was included (though not compiled in by default) in distributions prior to versions 1.2 as the ***mod_fastcgi*** module. It is not included now due to the problem of synchronizing versions. Commercial implementations of FastCGI are available for Netscape servers and Microsoft IIS from Fast Engines, Inc.). Fast.Serv is another commerical implementation of FastCGI and is currently available for all Netscape and Microsoft Web servers on Windows NT and all major UNIX platforms.

More information on FastCGI, including FastCGI server modules and application libraries, is available at FastCGI Web site.

The limitations of FastCGI are:

- FastCGI does not work natively with some Web servers and requires a specific add-on which can reduce performance advantage.
- FastCGI has the problem of process proliferation: there is at least one process for each FastCGI program. It needs to switch context to another heavyweight process.
- If a FastCGI program is to handle concurrent requests, it needs a pool of processes, one per request. If each of these requests is executing a Perl interpreter, this approach does not scale well. (This problem can be circumvented somewhat as FastCGI can distribute its processes across multiple servers, but then that requires extra resources.)
- FastCGI, like CGI, does not interact with the Web server.
- FastCGI programs are only as portable as the language they are written in.

2.9 Preprocessing and Caching

If the number of possible inputs and state combinations is small, one can run the CGI for all possible input offline and cache each result in a static HTML document.The limitation to this approach is that it may not work if the browser does not cache documents. Also, there are cases such as outputs of CGI scripts, which should not be cached. In such cases, the scripts need to specify the appropriate header (***Pragma:Nocache*** in HTTP 1.0 or ***Cache-Control*** in HTTP 1.1), and as a result put load on the server.

2.9.1 Server Redirection

When CGI scripts retrieve and return an ***existing*** document (on any server), it is known as server-redirection. It can be done using the HTTP ***Location:*** response header pointing to the static HTML document. In Apache, you can also redirect an entire server or

directory to a single URL using the Apache module ***mod_rewrite***. (The CGI approach for redirection is preferred if any information is being POSTed to the redirected URL.)
Server-redirection can have various applications, such as, returning a standard response page when a user submits a feedback form. When there are large number of inputs but a small number of frequently requested documents, caching is possible via server-redirection.

2.10 Embedded Interpreters

A solution to the CGI performance problem is using embedded high-level interpretive languages in their servers. Embedded interpreters often come with CGI emulation layers, allowing scripts to be executed directly by the server without the overhead of invoking a separate process. An embedded interpreter also eliminates the need to make dramatic changes to the server software itself. In many cases (and in contrast to server proprietary APIs), an embedded interpreter provides a smooth path for speeding-up CGI scripts because little or no source code modification is necessary.

2.10.1 mod_perl

One of the most important developments (and natural choice both from the language and the server standpoint) in the embedded interpreter arena has been the provison of including a Perl interpreter within the Apache Web server.
mod_perl is an Apache server module that embeds a copy of the Perl interpreter into the server executable. With *mod_perl*, Perl becomes the extension language for the Web server, providing a complete access to the Perl functionality within Apache. One can then write Perl snippets or CGI scripts, which ***do not*** require a new Perl interpreter process to be invoked (since Perl is not built-in the server). Instead, a new thread executes a precompiled Perl program. Since the CGI scripts (in Perl) are precompiled by the server and executed without forking, they running more quickly and efficiently. (Usually, it is not the size of the script itself but the *fork/exec* overhead that slows a CGI down.)
For Web servers under Windows NT there are other solutions. PerlScript is an ActiveX scripting engine that lets you embed Perl code in you Web pages (in a manner similar to JavaScript or VBScript). PerlIS is an ISAPI DLL that runs Perl scripts directly from Microsoft IIS and other ISAPI-compliant Web servers, providing significant benefits.

Both these solutions are from ActiveWare, which also provide prebuilt Perl binaries for Windows 9x/NT.

2.11 Goodbye to Performance

Last but not least, it almost goes without saying that the program scripts should be kept as small as possible (but not smaller). It has various advantages such as ease of testing, debugging and maintainance. These factors are directly/indirectly related to performance. Optimization in code *size* also means using a context dependent approach and avoiding "overkill." The moral: "Keep It Small, Silly" (or say goodbye to performance).

2.12 Conclusion

CGI is inflicted with various limitations. However, some of these can be circumvented just by careful scripting, with using the strengths and knowing the weaknesses of the language of choice.

There are other (server-side) alternatives to CGI for creating dynamic content, such as serve extension APIs, Servlets, Active Server Pages (ASP), Server-Side JavaScript and enhancements to Server-Side Includes. Some of these avoid almost all the problems inherent in the CGI but come with other trade-offs.