

The ApacheCon 2000
March 9, 2000
Orlando, Florida

**Improving Script Performance Under
mod_perl**

By Stas Bekman
Internet and Intranet programmer
<http://stason.org/>
<stas@stason.org>

This document is originally written in **POD**, converted to **HTML** by **pod2html** utility and then to **PostScript** by **html2ps** utility.

Copyright © 1998, 1999 Stas Bekman. All rights reserved.

(you will find a Table of Contents at the end)

1 Getting Started Fast

1.1 mod_perl in Four Slides

Each tutorial will concentrate on different aspects of running a mod_perl server and mod_perl programming. In case you don't know how to get started with it, or you think it's a difficult task, these slides will take away any worries you might have had when you came to this tutorial.

In just four slides you will be able to install and configure a mod_perl server. And, of course, to write new code and reuse the existing code under mod_perl.

The four slides (sections) are:

- Installation
- Configuration
- The “mod_perl rules” Apache::Registry Scripts
- The “mod_perl rules” Apache Perl Module

1.2 What is mod_perl?

But before we go any further, there is a chance that you don't know what mod_perl is. So let's make a little introduction to mod_perl.

Everybody knows that Perl scripts running under mod_cgi have numerous shortcomings. There are many of them, but code recompilation and Perl interpreter loading overhead at each request is the hardest one to overcome.

Among various attempts to improve on mod_cgi's shortcomings, mod_perl has proved to be one of the better ones and has been widely adopted by CGI developers. According to the <http://perl.apache.org/netcraft/> about 412000 hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the “Artistic License” as Perl itself.

mod_perl does away with mod_cgi's forking by reusing the existing child processes. In this new model, the child process doesn't exit anymore when it has processed a request. The Perl interpreter is loaded only once, when the process is started. Since the interpreter is persistent throughout the process' lifetime, all code is loaded and compiled only once, the first time it is seen. This makes all subsequent requests run much faster because everything is already loaded and compiled. Response processing is now reduced to running your code. This improves response times by a factor of 10 to 100, depending on the code being executed.

Doug didn't stop here, he went and extended mod_cgi's functionality by adding a complete Perl API to the Apache core. This makes it possible to write a complete Apache module in Perl, a feat that used to require coding in C. From then on mod_perl enabled the programmer to handle all phases of request processing in Perl.

The new Perl API also allows complete server configuration in Perl. This has which made the lives of many server administrators much easier, as they could now benefit from dynamically generating the configuration, freed from hunting for bugs in huge configuration files full of similar directives for virtual hosts and the like.

To provide backwards compatibility for plain CGI scripts that used to be run under `mod_cgi`, while still benefiting from a preloaded perl and modules, a few special handlers were written, each allowing a different level of proximity to pure `mod_perl` functionality. Some take full advantage of `mod_perl`, while others only a partial one.

`mod_perl` embeds a copy of the Perl interpreter into the Apache `httpd` executable, providing complete access to Perl functionality within Apache. This enables a set of `mod_perl`-specific configuration directives, all of which start with the string `Perl*`. Most, but not all, of these directives are used to specify handlers for various phases of the request.

It might occur to you that sticking a large executable (Perl) into another large executable (Apache) makes a very, very large program. `mod_perl` certainly makes `httpd` significantly bigger and you will need more RAM on your production server to be able to run many `mod_perl` processes, but in reality the situation is different. Since `mod_perl` processes requests much faster, the number of the processes needed to handle the same request rate is much lower relative to the `mod_cgi` approach. Generally you need slightly more memory available, and the speed improvements you will see are well worth every megabyte of memory you can add.

Now let's get back to the *All-In-Four-Slides...*

1.3 Installation

Did you know that it takes about 10 minutes to build and install a `mod_perl` enabled Apache server on a computer with a pretty average processor and a decent amount of system memory? It goes like this:

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

That's all!

- Of course you must replace `x.x.x` with the actual version numbers of the `mod_perl` and Apache releases that you use.

- The GNU `tar` utility knows how to uncompress a gzipped tar archive (use the `z` option).

All that's left is to add a few configuration lines to a *httpd.conf*, an Apache configuration file, start the server and enjoy `mod_perl`.

1.4 Configuration

Add the following to the configuration file *httpd.conf*:

```
# for Apache::Registry mode
Alias /perl/ /home/httpd/perl/

PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

This configuration causes every URI starting with */perl* to be handled by the Apache `mod_perl` module. It will use the handler from the Perl module `Apache::Registry`.

1.5 The "mod_perl rules" Apache::Registry Scripts

You can write plain perl/CGI scripts just as under `mod_cgi`:

```
mod_perl_rules1.pl
-----
print "Content-type: text/plain\r\n\r\n";
print "mod_perl rules!\n";
```

Of course you can write them in the Apache Perl API:

```
mod_perl_rules2.pl
-----
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

Save both files under the */home/httpd/perl* directory, make them executable and readable by server, and issue these requests using your favorite browser:

```
http://localhost/perl/mod\_perl\_rules1.pl
http://localhost/perl/mod\_perl\_rules2.pl
```

In both cases you will see on the following response:

```
mod_perl rules!
```

1.6 The "mod_perl rules" Apache Perl Module

To create an Apache Perl module, all you have to do is to wrap the code into a handler subroutine and return the status to the server.

```
ModPerl/Rules.pm
-----
use Apache::Constants;

sub handler{
    my $r = shift;
    $r->send_http_header('text/plain');
    print "mod_perl rules!\n";
    return OK;
}
```

Create a directory called *ModPerl* under one of the directories in @INC, and put *Rules.pm* into it. Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules
<Location /mod_perl_rules>
    SetHandler perl-script
    PerlHandler ModPerl::Rules
</Location>
```

Now you can issue a request to:

```
http://localhost/perl/mod\_perl\_rules
```

and just as with our *mod_perl_rules.pl* scripts you will see:

```
mod_perl rules!
```

as the response.

1.7 Is That All I Need To Know About mod_perl?

Definitely not!

These slides are intended to show you that you can install and start using a mod_perl server within 30 minutes of downloading the sources.

There is much more to mod_perl than this, you will need to plan your study around the projects you want to implement. Fortunately, there are many resources and lots of help freely available to you.

At the end of each tutorial you will find a chapter describing the available resources and pointers to them.

;o)

2 Performance. Benchmarks.

2.1 What we will learn in this chapter

- Performance: An Overall picture
- Analysis of SW and HW Requirements
- Sharing Memory
- How Shared My Memory Is
- Preload Perl modules at server startup
- Preload Registry Scripts
- Global vs Fully Qualified Variables
- Avoid Importing Functions
- PerlSetupEnv Off
- Adding a Proxy Server in http Accelerator Mode
- KeepAlive
- Upload/Download of Big Files
- Forking or Executing subprocesses from mod_perl
- Memory leakage
- Checking script modification times
- Cached `stat()` calls
- Be carefull with symbolic links
- Limiting the size of the processes
- Limiting the resources used by httpd children
- Limiting the request rate speed (robots blocking)
- Benchmarks. Impressing your Boss and Colleagues.
- Tuning the Apache's configuration variables for the best performance
- Persistent DB Connections

- Using `$|=1` under mod_perl and better `print()` techniques.
- Object Methods Calls Versus Function Calls
- Sending plain HTML as a compressed output

2.2 Performance: An Overall picture

Before we dive into performance issues, there is something very important to understand. It applies to any webserver, not only apache. All the efforts are made to make user's web browsing experience a swift. Among other web site usability factors, speed is one of the most crucial ones. What is a correct speed measurement? Since user is the one that interacts with web site, speed measurement is a time passed from the moment user follows a link or presses a submit button till the resulting page is being rendered by her browser. So if we trace the data packet's movement as it leaves user's machine (request sent) till the reply arrives, the packet travels through many entities on its way. It has to make its way through the network, passing many interconnection nodes, before it enters the target machine it might go through proxy (accelerator) servers, then it's being served by your server, and finally it has to make the whole way back. A webserver is only one of the elements the packet sees on its way. You could work hard to fine tune your webserver for the best performance, but a slow NIC (Network Interface Card) or slow network connection from your server might defeat it all. That's why it's important to think big and to be aware of possible bottlenecks between the server and the web. Of course there is nothing you can do if user has a slow connection on its behalf.

Moreover, you might tune your scripts and webserver to process incoming requests ultra fast, so you will need a little number of working servers, but you might find out that server processes are busy waiting for slow clients to complete the download. You will see more examples in this chapter.

My point is that a web service is like car, if one of the details or mechanisms is broken the car will not drive smoothly and it can even stop dead if pushed further without first fixing it.

2.3 Analysis of SW and HW Requirements

You need to analyze all of the problem's dimensions. There are several things that need to be considered:

- *How long does it take to process each request
- *How many requests can you process simultaneously
- *How many simultaneous requests are you planning to get

The first one is probably the easiest to optimize. Follow the performance optimization tips in the guide and other docs, let a professional perl (mod_perl) programmer to work out your code and improve it.

The second one is a function of RAM. How much RAM is in the box, how many boxes do you have, and how much RAM does each mod_perl process take? Multiply the first two and divide by the third. Ask yourself whether it is better to switch to another, possibly just as inefficient language will actually cost more than throwing another Ultra 2 into the rack. Also ask yourself whether switching to another language

will even help. In some applications, a huge chunk of memory is needed e.g. to link in Oracle runtime libraries. So you would pay this price even if you switch from Perl to C.

The last one is important. You need to have a realistic answer. Are you really expecting 8 million hits per day? What is the expected peak load, and what kind of response time do you need to guarantee? Remember that this numbers might change drastically when you apply code changes and your site becomes more popular. Remember that when the you get a very high hits rate, the requirements wouldn't grow lineary by exponentialy!

2.4 Sharing Memory

A very important point is the sharing of memory. If your OS supports this (and most sane systems do), you might save more memory by sharing it between child processes. This is only possible when you preload code at server startup. However during a child process' life, its memory pages becomes unshared and there is no way we can control perl to make it allocate memory so (dynamic) variables land on different memory pages than constants, that's why the **copy-on-write** effect (will explain in a moment) will hit almost at random. If you are pre-loading many modules you might be able to balance the memory that stays shared against the time for an occasional fork by tuning the `MaxRequestsPerChild` to a point where you restart before too much becomes unshared. In this case the `MaxRequestsPerChild` is very specific to your scenario. You should do some measurements and you might see if this really makes a difference and what a reasonable number might be. Each time a child reaches this upper limit and restarts it should release the unshared copies and the new child will inherit pages that are shared until it scribbles on them.

It is very important to understand that your goal is not to have `MaxRequestsPerChild` to be 10000. Having a child serving 300 requests on precompiled code is already a huge speedup, so if it is 100 or 10000 it does not really matter if it saves you the RAM by sharing. Do not forget that if you preload most of your code at the server startup, the fork to spawn a new child will be very very fast, because it inherits most of the preloaded code and the perl interpreter from the parent process. But than, during the work of the child, its memory pages (which aren't really its yet, it uses the parent's pages) are getting dirty (originally inherited and shared variables are getting updated/modified) and the **copy-on-write** happens, which reduces the number of shared memory pages - thus enlarging the memory demands. Killing the child and respawning a new one, allows to get the pristine shared memory from the parent process again.

The conclusion is that `MaxRequestsPerChild` should not be too big, otherwise you loose the benefits of the memory sharing.

2.5 How Shared My Memory Is

You've probably noticed that the word shared is being repeated many times in many things related to `mod_perl`. Indeed, shared memory might save you a lot of money, since with sharing in place you can run many more servers than without it.

How much shared memory do you have? You can see it by either using the memory utils that comes with your system or you can deploy `GTop` module:

```
print "Shared memory of the current process: ",
      GTop->new->proc_mem($$)->share, "\n";
```

```
print "Total shared memory: ",
      GTop->new->mem->share, "\n";
```

When you watch the output of the `top` utility, don't confuse **RSS** (or **RES**) column with **SHARE** column -- **RES** is a RESident memory, which is a size of pages currently swapped in.

2.6 Preload Perl modules at server startup

Use the `PerlRequire` and `PerlModule` directives to load commonly used modules such as `CGI.pm`, `DBI` and etc., when the server is started. On most systems, server children will be able to share the code space used by these modules. Just add the following directives into `httpd.conf`:

```
PerlModule CGI;
PerlModule DBI;
```

But even a better approach is to create a separate startup file (where you code in plain perl) and put there things like:

```
use DBI;
use Carp;
```

Then you `require()` this startup file with help of `PerlRequire` directive from `httpd.conf`, by placing it before the rest of the `mod_perl` configuration directives:

```
PerlRequire /path/to/start-up.pl
```

`CGI.pm` is a special case. Ordinarily `CGI.pm` autoloads most of its functions on an as-needed basis. This speeds up the loading time by deferring the compilation phase. However, if you are using `mod_perl`, `FastCGI` or another system that uses a persistent Perl interpreter, you will want to precompile the methods at initialization time. To accomplish this, call the package function `compile()` like this:

```
use CGI ();
CGI->compile(':all');
```

The arguments to `compile()` are a list of method names or sets, and are identical to those accepted by the `use()` and `import()` operators. Note that in most cases you will want to replace `:all` with tag names you really use in your code, since generally only a subset of subs is actually being used.

2.6.1 Preload Perl modules - Real Numbers

I have conducted a few tests to benchmark the memory usage when some modules are preloaded. The first set of tests checks the memory use with Library Perl Module preload (only `CGI.pm`). The second set checks the `compile` method of `CGI.pm`. The third test checks the benefit of Library Perl Module preload but a few of them (to see more memory saved) and also the effect of precompiling the Registry modules with `Apache::RegistryLoader`.

1. In the first test, the following script was used:

```
use strict;
use CGI ();
my $q = new CGI;
print $q->header;
print $q->start_html,$q->p("Hello");
```

Server restarted

Before the CGI .pm preload: (No other modules preloaded)

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	87004	0.0	0.0	1060	1524	-	A	16:51:14	0:00	httpd
httpd	240864	0.0	0.0	1304	1784	-	A	16:51:13	0:00	httpd

After running a script which uses CGI's methods (no imports):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	188068	0.0	0.0	1052	1524	-	A	17:04:16	0:00	httpd
httpd	86952	0.0	1.0	2520	3052	-	A	17:04:16	0:00	httpd

Observation: child httpd has grown up by 1268K

Server restarted

After the CGI .pm preload:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	240796	0.0	0.0	1456	1552	-	A	16:55:30	0:00	httpd
httpd	86944	0.0	0.0	1688	1800	-	A	16:55:30	0:00	httpd

after running a script which uses CGI's methods (no imports):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86872	0.0	0.0	1448	1552	-	A	17:02:56	0:00	httpd
httpd	187996	0.0	1.0	2808	2968	-	A	17:02:56	0:00	httpd

Observation: child httpd has grown up by 1168K, 100K less then without preload - good!

Server restarted

After CGI .pm preloaded and compiled with CGI->compile(':all');

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86980	0.0	0.0	2836	1524	-	A	17:05:27	0:00	httpd
httpd	188104	0.0	0.0	3064	1768	-	A	17:05:27	0:00	httpd

After running a script which uses CGI's methods (no imports):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86980	0.0	0.0	2828	1524	-	A	17:05:27	0:00	httpd
httpd	188104	0.0	1.0	4188	2940	-	A	17:05:27	0:00	httpd

Observation: child httpd has grown up by 1172K No change! So what does CGI->compile(':all') help? I think it's because we never use all of the methods CGI provides - so in real use it's faster. So you might want to compile only the tags you are about to use - then you will benefit for sure.

2. I have tried the second test to find it. I run the script:

```
use strict;
use CGI qw(:all);
print header,start_html,p("Hello");
```

Server restarted

After CGI .pm was preloaded and NOT compiled with CGI->compile(':all'):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	17268	0.0	0.0	1456	1552	-	A	18:02:49	0:00	httpd
httpd	86904	0.0	0.0	1688	1800	-	A	18:02:49	0:00	httpd

After running a script which imports symbols (all of them):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	17268	0.0	0.0	1448	1552	-	A	18:02:49	0:00	httpd
httpd	86904	0.0	1.0	2952	3112	-	A	18:02:49	0:00	httpd

Observation: child httpd has grown up by 1264K

Server restarted

After CGI .pm was preloaded and compiled with CGI->compile(':all'):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86812	0.0	0.0	2836	1524	-	A	17:59:52	0:00	httpd
httpd	99104	0.0	0.0	3064	1768	-	A	17:59:52	0:00	httpd

After running a script which imports symbols (all of them):

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	86812	0.0	0.0	2832	1436	-	A	17:59:52	0:00	httpd
httpd	99104	0.0	1.0	4884	3636	-	A	17:59:52	0:00	httpd

Observation: child httpd has grown up by 1868K. Why? Isn't CGI::compile(':all') supposed to make children to share the compiled code with parent? It does work as advertised, but if you pay attention in the code we have called only three CGI.pm's methods - just saying use CGI qw(:all) doesn't mean we compile the all available methods - we just import their names. So actually this test is misleading. Execute compile() only on the methods you are actually using and then you will see the difference.

3. The third script:

```

use strict;
use CGI;
use Data::Dumper;
use Storable;
[and many lines of code, lots of globals - so the code is huge!]

```

Server restarted

Nothing preloaded at startup:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	90962	0.0	0.0	1060	1524	-	A	17:16:45	0:00	httpd
httpd	86870	0.0	0.0	1304	1784	-	A	17:16:45	0:00	httpd

Script using CGI (methods), Storable, Data::Dumper called:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	90962	0.0	0.0	1064	1436	-	A	17:16:45	0:00	httpd
httpd	86870	0.0	1.0	4024	4548	-	A	17:16:45	0:00	httpd

Observation: child httpd has grown by 2764K

Server restarted

Preloaded CGI (compiled), Storable, Data::Dumper at startup:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	26792	0.0	0.0	3120	1528	-	A	17:19:21	0:00	httpd
httpd	91052	0.0	0.0	3340	1764	-	A	17:19:21	0:00	httpd

Script using CGI (methods), Storable, Data::Dumper called

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	26792	0.0	0.0	3124	1440	-	A	17:19:21	0:00	httpd
httpd	91052	0.0	1.0	6568	5040	-	A	17:19:21	0:00	httpd

Observation: child httpd has grown by 3276K. Ouch: 512K more!!!

The reason is that when you preload at the startup all of the methods, they all are being precompiled, there are many of them and they take a big chunk of memory. If you don't use the `compile()` method, only the functions that are being used will be compiled. Yes, it will slightly slow down the first response of each process, but the actual memory usage will be lower. BTW, if you write in the script:

```
use CGI qw(all);
```

Only the symbols of all functions are being imported. While they are taking some space, it's smaller than the space that a compiled code of these functions might occupy.

Server restarted

All the above modules + the above script PreCompiled with `Apache::RegistryLoader` at startup:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	43224	0.0	0.0	3256	1528	-	A	17:23:12	0:00	httpd
httpd	26844	0.0	0.0	3488	1776	-	A	17:23:12	0:00	httpd

Script using CGI (methods), `Storable`, `Data::Dumper` called:

USER	PID	%CPU	%MEM	SZ	RSS	TTY	STAT	STIME	TIME	COMMAND
root	43224	0.0	0.0	3252	1440	-	A	17:23:12	0:00	httpd
httpd	26844	0.0	1.0	6748	5092	-	A	17:23:12	0:00	httpd

Observation: child httpd has grown even more 3316K ! Does not seem to be good!

Summary:

1. Library Perl Modules Preloading gave good results everywhere.
2. CGI.pm's `compile()` method seems to use even more memory. It's because we never use all of the methods CGI provides. Do `compile()` only the tags that you are going to use and you will save the overhead of the first call for each has not yet been called method, and the memory - since compiled code will be shared across all the children.
3. `Apache::RegistryLoader` might make scripts load faster on the first request after the child has just started but the memory usage is worse!!! See the numbers by yourself.

HW/SW used : The server is apache 1.3.2, mod_perl 1.16 running on AIX 4.1.5 RS6000 1G RAM.

2.7 Preload Registry Scripts

`Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup. It can be a good idea to preload the scripts you are going to use as well. So the code will be shared among the children.

Here is an example of the use of this technique. This code is included in a `PerlRequire`'d file, and walks the directory tree under which all registry scripts are installed. For each `.pl` file encountered, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server (before pre-forking the child processes):

```
use File::Find 'finddepth';
use Apache::RegistryLoader ();
{
    my $perl_dir = "perl/";
    my $rl = Apache::RegistryLoader->new;
    finddepth(sub {
        return unless /\.pl$/;
        my $url = "$File::Find::dir/$_";
        print "pre-loading $url\n";

        my $status = $rl->handler($url);
        unless($status == 200) {
```

```

        warn "pre-load of '$url' failed, status=$status\n";
    }
}, $perl_dir);
}

```

Note that we didn't use the second argument to `handler()` here, as module's manpage suggests. To make the loader smarter about the `uri->filename` translation, you might need to provide a `trans()` function to translate the `uri` to `filename`. URI to filename translation normally doesn't happen until HTTP request time, so the module is forced to roll its own translation. If `filename` is omitted and a `trans()` routine was not defined, the loader will try using the URI relative to **ServerRoot**.

2.8 Global vs Fully Qualified Variables

It's always a good idea to stay away from global variables when possible. Some variables must be global so Perl can see them, such as a module's `@ISA` or `$VERSION` variables (or fully qualified **@MyModule::ISA**). In common practice, a combination of `strict` and `vars` pragmas keeps modules clean and reduces a bit of noise. However, `vars` pragma also creates aliases as the `Exporter` does, which eat up more memory. When possible, try to use fully qualified names instead of use `vars`. Example:

```

package MyPackage;
use strict;
@MyPackage::ISA = qw(...);
$MyPackage::VERSION = "1.00";

```

vs.

```

package MyPackage;
use strict;
use vars qw(@ISA $VERSION);
@ISA = qw(...);
$VERSION = "1.00";

```

2.9 PerlSetupEnv Off

`PerlSetupEnv Off` is another optimization you might consider.

`mod_perl` fiddles with the environment to make it appear as if the script were being called under the CGI protocol. For example, the `$ENV{QUERY_STRING}` environment variable is initialized with the contents of `Apache::args()`, and `$ENV{SERVER_NAME}` is filled in from the value returned by `Apache::server_hostname()`.

But `%ENV` population is expensive. Those who have moved to the Perl Apache API no longer need this extra `%ENV` population, can gain by turning it **Off**.

By default it is On.

Note that you can still set ENV variables. e.g. when you use the following configuration:

```
<Location /perl>
  PerlSetupEnv Off
  PerlSetEnv TEST hi
  SetHandler perl-script
  PerlHandler Apache::RegistryNG->handler
  Options +ExecCGI
</Location>
```

A script having a `print Data::Dumper(\%ENV)` line, prints:

```
$VAR1 = {
  'GATEWAY_INTERFACE' => 'CGI-Perl/1.1',
  'MOD_PERL' => 'mod_perl/1.21_01-dev',
  'PATH' => '/usr/lib/perl5/5.00503:... snipped ...',
  'TEST' => 'hi'
};
```

2.10 Adding a Proxy Server in http Accelerator Mode

At the beginning there were 2 servers: one plain apache server, which was *very light*, and configured to serve static objects, the other mod_perl enabled (*very heavy*) and configured to serve mod_perl scripts. We named them `httpd_docs` and `httpd_perl` respectively.

The two servers coexist at the same IP address by listening to different ports: `httpd_docs` listens to port 80 (e.g. <http://www.nowhere.com/images/test.gif>) and `httpd_perl` listens to port 8080 (e.g. <http://www.nowhere.com:8080/perl/test.pl>). Note that I did not write <http://www.nowhere.com:80> for the first example, since port 80 is the default port for the http service. Later on, I will be changing the configuration of the `httpd_docs` server to make it listen to port 81.

Now I am going to convince you that you **want** to use a proxy server (in the http accelerator mode). The advantages are:

- Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the `httpd_docs` server).
- You get less I/O activity reading static objects from the disk (proxy serves the most "popular" objects from RAM - of course you benefit more if you allow the proxy server to consume more RAM). Since you do not wait for the I/O to be completed you are able to serve static objects much faster.
- The proxy server acts as a sort of output buffer for the dynamic content. The mod_perl server sends the entire response to the proxy and is then free to deal with other requests. The proxy server is responsible for sending the response to the browser. So if the transfer is over a slow link, the mod_perl server is not waiting around for the data to move.

Using numbers is always more convincing :) Let's take a user connected to your site with 28.8 kbps (bps == bits/sec) modem. It means that the speed of the user's link is $28.8/8 = 3.6$ kbytes/sec. I assume an average generated HTML page to be of 10kb (kb == kilobytes) and an average script that generates this output in 0.5 secs. How long will the server wait before the user gets the whole output response? A simple calculation reveals pretty scary numbers - it will have to wait for another 6 secs

(20kb/3.6kb), when it could serve another 12 (6/0.5) dynamic requests in this time.

This very simple example shows us that we need only one twelfth the number of children running, which means that we will need only one twelfth of the memory (not quite true because some parts of the code are shared).

But you know that nowadays scripts often return pages which are blown up with javascript code and similar, which can make them of 100kb size and the download time will be of the order of... (This calculation is left to you as an exercise :)

Many users like to open many browser windows and do many things at once (download files and browse graphically *heavy* sites). So the speed of 3.6kb/sec we were assuming before, may often be 5-10 times slower.

- We are going to hide the details of the server's implementation. Users will never see ports in the URLs (more on that topic later). You can have a few boxes serving the requests, and only one serving as a front end, which spreads the jobs between the servers in a way that you can control. You can actually shut down a server, without the user even noticing, because the front end server will dispatch the jobs to other servers. (This is called a Load Ballancing and it's a pretty big issue, which will not be discussed in this document.)
- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever. The httpd accelerator and internal server communicate in expected HTTP requests. This allows for only your public "bastion" accelerating www server to get hosed in a successful attack, while leaving your internal data safe.

The disadvantages are:

- Of course there are drawbacks. Luckily, these are not functionality drawbacks, but they are more administration hassle. You have another daemon to worry about, and while proxies are generally stable, you have to make sure to prepare proper startup and shutdown scripts, which are run at boot and reboot as appropriate. Also, you might want to set up the crontab to run a watchdog script.
- Proxy servers can be configured to be light or heavy, the admin must decide what gives the highest performance for his application. A proxy server like squid is light in the concept of having only one process serving all requests. But it can appear pretty heavy when it loads objects into memory for faster service.

If you are on a local area network (LAN), then the big benefit of the proxy buffering the output and feeding a slow client is gone. You are probably better off sticking with a straight mod_perl server in this case.

2.11 KeepAlive

If your mod_perl server's *httpd.conf* includes the following directives:

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
```

you've gotten a real performance penalty, since after completing each request processing, the process will wait for `KeepAliveTimeout` seconds before closing the connection and thus not serving other requests at this time. You will need many more processes on a server with high traffic.

If you use some server status reporting tools, you will see the process in *K* status when it's in `KeepAlive` status.

Most chances are that you don't want this feature to be enabled. So set it `Off` with:

```
KeepAlive Off
```

the other two directive don't matter anymore.

You might want to consider to enable this option if the client's browser needs to bring more than one object from your server at once (for a single HTML page). If this is the situation you actually save the connection overhead for all requests but the first one.

For example if you have a page with 10 ad banners, which is not uncommon today, your server will work more effectively if a single process will serve them all during a single connection. Your client will get a little slower response, since banners will be brought one at a time and not all together if each `IMG` tag would open a separate connection.

There are definite advantages to keep-alive from a TCP perspective since fresh connections will incur not only the 3 way-TCP handshake but also be penalised by slow-start. So while turning it off may help the memory usage on the server, it will disadvantage the client from a network speed perspective.

You probably have followed the advice of sending all the static object requests to a plain Apache server. And since most of the pages include more than one static unique image, you better keep the default setting of the non-mod_perl server, which has the `KeepAlive` directive `On`. Probably reducing a little the number of timeout seconds is a good idea too.

One option I suppose would be for the proxy/accelerator to keep the connection open to the client but make individual connections to the server, read the response, buffer it for sending to the client and close the server connection (making new connections to the server as required by the client requests obviously).

2.12 Upload/Download of Big Files

If some particular script's main functionality is uploading or downloading of big files, you probably want it to be executed on plain apache server under `mod_cgi`. Taken of course that the script requires none of the functionalities the mod_perl server provides. Like custom authentication handlers.

You don't want to tie up your precious mod_perl backend server children doing something as long and dumb as transferring a file.

Also, the user won't really see any important performance benefits from mod_perl anyway, since the upload may take up to several minutes, and the overhead saved by mod_perl is typically under one second.

2.13 Forking or Executing subprocesses from mod_perl

Generally you should not fork from your mod_perl scripts, since when you do -- you are forking the entire apache web server, lock, stock and barrel. Not only is your perl code being duplicated, but so is mod_ssl, mod_rewrite, mod_log, mod_proxy, mod_spelling or whatever modules you have used in your server, all the core routines and so on.

A much wiser approach would be to spawn a sub-process, hand it the information it needs to do the task, and have it detach (`close x3 + setsid()`). This is wise only if the parent who spawns this process, immediately continue, you do not wait for the sub-process to complete. This approach is suitable for a situation when you want to trigger a long time taking process through the web interface, like processing some data, sending email to thousands of subscribed users and etc. Otherwise, you should convert the code into a module, and use its functions or methods to call from CGI script.

Just making a `system()` call defeats the whole idea behind mod_perl, perl interpreter and modules should be loaded again for this external program to run.

Basically, you would do:

```
$params=FreezeThaw::freeze(
    [all data to pass to the other process]
);
system("program.pl $params");
```

and in `program.pl` :

```
use POSIX qw(setsid);
@params=FreezeThaw::thaw(shift @ARGV);
# check that @params is ok
close STDIN;
close STDOUT;
close STDERR;
# you might need to reopen the STDERR
# open STDERR, ">/dev/null";
setsid(); # to detach
```

At this point, `program.pl` is running in the "background" while the `system()` returns and permits apache to get on with life.

This has obvious problems. Not the least of which is that `@params` must not be bigger than whatever your architecture's limit is (could depend on your shell).

Also, the communication is only one way.

However, you might want be trying to do the “wrong thing”. If what you want is to send information to the browser and then do some post-processing, look into `PerlCleanupHandler`.

If you are interested in more deep level details, this is what actually happens when you fork and make a system call, like

```
system("echo Hi"),CORE::exit(0) unless fork();
```

which is might be more familiar in this form:

```
if (fork){
    #do nothing
} else {
    system("echo Hi");
    CORE::exit(0);
}
```

What happens is that `fork()` gives you 2 execution paths and the child gets virtual memory sharing a copy of the program text (read only) and sharing a copy of the data space copy-on-write (remember why you pre-load modules in `mod_perl`?). In the above code a parent will immediately continue with the code that comes up after the fork, while the forked process will execute `system("echo Hi")` and then terminate itself.

Notice that I use `CORE::exit` and not `exit` which would be automatically overridden by `Apache::exit` if used in conjunction with `Apache::Registry` and friends.

The only work is setting up the page tables for the virtual memory and the second process goes on its separate way.

Next, Perl will find `/bin/echo` along the search path, and invoke it directly. Perl `system()` is **not** `system(3)` [C-library]. Only when the command has shell meta-chars does Perl invoke a real shell. That's a **very** nice optimization.

Only if you do:

```
system "sh -c 'echo foo'"
```

OS actually parses your command with a shell so you `exec()` a copy of `/bin/sh`, but since one is almost certainly already running somewhere, the system will notice that (via the disk inode reference) and replace your virtual memory page table with one pointed at the already-loaded program code plus your own data space. Then the shell parses the passed command.

Since it is `echo`, it will execute it as a built-in in the latter example or a `/bin/echo` in the former and be done, but this is only an example. You aren't calling `system("echo Hi")` in your `mod_perl` scripts, right? Since most other real things (heavy programs executed as a subprocess) would involve repeating the process to load the specified command or script (it might involve some actual demand paging from the program file if you execute new code).

The only place you see real overhead from this scheme is when the parent process is huge (unfortunately like mod_perl...) and the page table becomes large as a side effect. The whole point of mod_perl is to avoid having to `fork()` / `exec()` something on every hit, though. Perl can do just about anything by itself. However, you probably won't get in trouble until you hit about 30 forks/sec on a so-so pentium.

Now let's get to the gory details of forking. Normally, every process has its parent. Many processes are children of the `init` process, whose `PID` equals to 1. When you fork a process you must `wait()` or `waitpid()` for it to finish. If you don't wait for it becomes a zombie.

Zombie, is a process that doesn't have a father. When the child quits, it reports the termination to his parent. If no one `wait()`s to collect the exit status of the child, it gets "confused" and becomes a ghost process, that can be seen, but not killed. It will be killed only when you stop the `httpd` process that spawned it! (generally `top()` / `ps()` utilities display these processes with `<defunc>` tag, and you will see an increment of the zombies counter reported when doing `top()`.) These zombie processes can take up system resources and are generally undesirable.

So the proper fork is:

```
print "Content-type: text/plain\n\n";

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
    print "Parent has finished\n";
} else {
    # do something
    CORE::exit(0);
}
```

But in most cases the only reason you would want to fork is when you need to spawn a process that would take a lot of time to complete. So if the server child that spawns this process has to wait for it to finish, you gained nothing. You cannot neither wait for its completion, nor continue because you will get yet another zombie process.

The simplest solution is to ignore your dead children (this doesn't work everywhere, however).

```
$SIG{CHLD} = IGNORE;
```

When you set `CHLD` signal handler to `IGNORE`, all the processes will be collected by the `init` process and prevent from them to become zombies.

Note, that you cannot localize this setting with `local()`. If you do, it wouldn't take the desired effect.

The other thing that you must do -- is to close all the pipes to the connection socket that were opened by the parent process (a `STDIN` and a `STDOUT`) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. You may need to close and reopen a `STDERR` filehandler (It's opened to append to the `error_log` file as inherited by parent, so chances are that you want it to leave untouched).

So now the code would look like:

```
print "Content-type: text/plain\n\n";

$SIG{CHLD} = IGNORE;

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
    print "Parent has finished\n";
} else {
    close STDIN;
    close STDOUT;
    close STDERR;
    # do something long lasting
    CORE::exit(0);
}
```

Another more portable, but slightly more expensive solution is to use a double fork approach.

```
print "Content-type: text/plain\n\n";

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
} else {
    defined (my $grandkid = fork) or die "Kid cannot fork: $!\n";
    if ($grandkid) {
        CORE::exit(0);

    } else {
        # code here
        close STDIN;
        close STDOUT;
        close STDERR;
        # do something long lasting
        CORE::exit(0);
    }
}
```

Grandkid becomes a *"child of init"* (parent process ID is 1).

Note that the last two solutions do allow you to know the exit status of the process, but in our case we don't want to.

One more solution is to use a different *SIGCHLD* handler:

```
use POSIX 'WNOHANG';
$SIG{CHLD} = sub { while( waitpid(-1,WNOHANG)>0 ) {} };
```

Which is useful when you `fork()` more than once process. The handler could call `wait()` as well, but for a variety of reasons involving the handling of stopped processes and the rare event in which two children exit at nearly the same moment, the best technique is to call `waitpid()` in a tight loop with a first argument of `-1` and a second argument of `WNOHANG`. Together these arguments tell `waitpid()` to reap

the next child that's available, and prevent the call from blocking if there happens to be no child ready from reaping. The handler will loop until `waitpid()` returns a negative number or zero, indicating that no more reapeable children remain.

You will probably want to open your own log file in the spawned process and log some info so you know what have happened there. At least while debugging your code.

Check also `Apache::SubProcess` for a better `system` and `exec` implementations for `mod_perl` (use CPAN!).

2.14 Memory leakage

Scripts under `mod_perl` can very easily leak memory! Global variables stay around indefinitely, lexical variables (declared with `my()`) are destroyed when they go out of scope, provided there are no references to them from outside of that scope.

Perl doesn't return the memory it acquired from the kernel. It does reuse it though!

First example demonstrates reading in a whole file:

```
open IN, $file or die $!;
local $/ = undef; # will read the whole file in
$content = <IN>;
close IN;
```

If your file is 5Mb, the child who served that script will grow exactly by that size. Now if you have 20 children and all of them will serve this CGI, all of them will consume additional $20 * 5M = 100M$ of RAM! If that's the case, try to use other approaches of processing the file, if possible of course. Try to process a line at a time and print it back to the file. (If you need to modify the file itself, use a temporary file. When finished, overwrite the source file, make sure to provide a locking mechanism!)

Second example demonstrates copying variables between functions (passing variables by value). Let's use the example above, assuming we have no choice but to read the whole file before any data processing takes place. Now you have some imagine `process()` subroutine that processes the data and returns it back. What happens if you pass the `$content` by value? You have just copied another 5M and the child has grown by another 5M in size (watch your swap space!) now multiply it again by factor of 20 you have 200M of wasted RAM, which will be apparently reused but it's a waste! Whenever you think the variable can grow bigger than few Kb, pass it by reference!

Once I wrote a script that passed a content of a little flat file DataBase to a function that processed it by value -- it worked and it was processed fast, but with a time the DataBase became bigger, so passing it by value was an overkill -- I had to make a decision, whether to buy more memory or to rewrite the code. It's obvious that adding more memory will be merely a temporary solution. So it's better to plan ahead and pass the variables by reference, if a variable you are going to pass might be bigger than you think at the time of your coding process. There are a few approaches you can use to pass and use variables passed by reference. For example:

```

my $content = qq{foobarfoobar};
process(\$content);
sub process{
    my $r_var = shift;
    $$r_var =~ s/foo/bar/g;
    # nothing returned - the variable $content outside has been
    # already modified
}

@{$var_lr} -- dereferences an array
%{$var_hr} -- dereferences a hash

```

For more info see `perldoc perlref`.

Another approach would be to directly use a `@_` array. Using directly the `@_` array serves the job of passing by reference!

```

process($content);
sub process{
    $_[0] =~ s/foo/bar/g;
    # nothing returned - the variable $content outside has been
    # already modified
}

```

From `perldoc perlsub`:

```

The array @_ is a local array, but its elements are aliases for
the actual scalar parameters. In particular, if an element
$_[0] is updated, the corresponding argument is updated (or an
error occurs if it is not possible to update)...

```

Be careful when you write this kind of subroutines, since it can confuse a potential user. It's not obvious that call like `process($content);` modifies the passed variable -- programmers (which are the users of your library in this case) are used to subs that either modify variables passed by reference or return the processed variable (e.g. `$content=process($content);`).

Third example demonstrates a work with DataBases. If you do some DB processing, many times you encounter the need to read lots of records into your program, and then print them to the browser after they are formatted. (I don't even mention the horrible case where programmers read in the whole DB and then use perl to process it!!! Use a relational DB and let the SQL do the job, so you get only the records you need!!!).

We will use DBI for this (assume that we are already connected to the DB) (refer to `perldoc DBI` for a complete manual of the DBI module):

```

$sth->execute;
while(@row_ary = $sth->fetchrow_array;) {
    <do DB accumulation into some variable>
}
<print the output using the the data returned from the DB>

```

In the example above the `httpd_process` will grow up by the size of the variables that have been allocated for the records that matched the query. (Again remember to multiply it by the number of the children your server runs!).

A better approach is to not accumulate the records, but rather print them as they are fetched from the DB. Moreover, we will use the `bind_col()` and `$sth->fetchrow_arrayref()` (aliased to `$sth->fetch()`) methods, to fetch the data in the fastest possible way. The example below prints a HTML TABLE with matched data, the only memory that is being used is a `@cols` array to hold temporary row values:

```
my @select_fields = qw(a b c);
    # create a list of cols values
my @cols = ();
@cols[0..$#select_fields] = ();
$sth = $dbh->prepare($do_sql);
$sth->execute;
    # Bind perl variables to columns.
$sth->bind_columns(undef,\(@cols));
print "<TABLE>";
while($sth->fetch) {
    print "<TR>",
        map("<TD>$_</TD>", @cols),
        "</TR>";
}
print "</TABLE>";
```

Note: the above method doesn't allow you to know how many records have been matched. The workaround is to run an identical query before the code above where you use `SELECT count(*) ...` instead of `'SELECT * ...` to get the number of matched records. It should be much faster, since you can remove any **SORTBY** and alike attributes.

2.15 Checking script modification times

Under `Apache::Registry` the requested CGI script is always being `stat()`'ed to check whether it was modified. It adds a very little overhead, but if you are into squeezing all the juices from the server, you might want to save this call. If you do -- take a look at `Apache::RegistryBB` module.

2.16 Cached stat() calls

When you do a `stat()` or its variations (`-M` - modification time, `-A` last access time, `-C` inode-change time, and other), the information is being cached, so if you need to make an additional check for the same file, save the overhead of this check and use a `_` variable instead. For example when testing for existence and read permissions you might use:

```
my $filename = "./test";
    # two stat() calls
print "OK\n" if -e $filename and -r $filename;
my $mod_time = (-M $filename) * 24 * 60 * 60;
print "$filename was modified $mod_time seconds ago\n";
```

or the more efficient (two `stat()` syscalls saved)!:

```
my $filename = "./test";
# two stat() calls
print "OK\n" if -e $filename and -r _;
my $mod_time = (-M _) * 24 * 60 * 60;
print "$filename was modified $mod_time seconds ago\n";
```

Remember that with `mod_perl` you might get negative times when you use `-M` and alike file tests. `-M` tests the difference in time between file modification file and the start of the script that performs this check. Because `^T` variable is not being reset on each script invocation, and equal to the time the process has been forked at, you might want to perform:

```
$$T = time();
```

at the beginning of your scripts to get the regular perl script behaviour of file tests

2.17 Be carefull with symbolic links

As you know `Apache::Registry` caches the scripts based on their URI. If you have the same script that can be reached by different URIs, possible if you have used a symbolic links, like:

```
% ln -s /home/httpd/perl/news/news.pl /home/httpd/perl/news.pl
```

Now the script can be reached as `/news/news.pl` and `/news.pl` URIs. It doesn't really matter until you advertise the two URIs, and users reach the same script from both of them. The moment this happens, you will get the same script cached twice!

Use `/perl-status` location handler to see all the compiled scripts and their packages. In our example when requesting: <http://localhost/perl-status?rgysubs> you would see:

```
Apache::ROOT::perl::news::news_2epl
Apache::ROOT::perl::news_2epl
```

after the both URIs have been requested from the same child process that happened to serve your request. To make the debug easier run the server in a single mode.

2.18 Limiting the size of the processes

`Apache::SizeLimit` allows you to kill off Apache httpd processes if they grow too large. see `perldoc Apache::SizeLimit` for more details.

By using this module, you should be able to discontinue using the Apache configuration directive `MaxRequestsPerChild`, although for some folks, using both in combination does the job.

2.19 Limiting the resources used by httpd children

`Apache::Resource` uses the `BSD::Resource` module, which uses the C function `setrlimit()` to set limits on system resources such as memory and cpu usage.

To configure use:

```
PerlModule Apache::Resource
# set child memory limit in megabytes
# (default is 64 Meg)
PerlSetEnv PERL_RLIMIT_DATA 32:48

# set child CPU limit in seconds
# (default is 360 seconds)
PerlSetEnv PERL_RLIMIT_CPU 120

PerlChildInitHandler Apache::Resource
```

If you configure `Apache::Status`, it will let you review the resources set this way.

The following limit values are in megabytes: `DATA`, `RSS`, `STACK`, `FSIZE`, `CORE`, `MEMLOCK`; all others are treated as their natural unit. Prepend `PERL_RLIMIT_` for each one you want to use. Refer to `setrlimit` man page on your OS for other possible resources.

If the value of the variable is of the form `S:H`, `S` is treated as the soft limit, and `H` is the hard limit. If it is just a single number, it is used for both soft and hard limits.

To debug add:

```
<Perl>
$Apache::Resource::Debug = 1;
require Apache::Resource;
</Perl>
PerlChildInitHandler Apache::Resource
```

and look in the `error_log` to see what it's doing.

Refer to `perldoc Apache::Resource` and `man 2 setrlimit` for more info.

2.20 Limiting the request rate speed (robots blocking)

A limitation of using pattern matching to identify robots is that it only catches the robots that you know about, and only those that identify themselves by name. A few devious robots masquerade as users by using user agent strings that identify themselves as conventional browsers. To catch such robots, you'll have to be more sophisticated.

`Apache::SpeedLimit` comes for you to help, see:

http://www.modperl.com/chapters/ch6.html#Blocking_Greedy_Clients

2.21 Benchmarks. Impressing your Boss and Colleagues.

How much faster is mod_perl than mod_cgi (aka plain perl/CGI)? There are many ways to benchmark the two. I'll present a few examples and numbers below. Checkout the benchmark directory of mod_perl distribution for more examples.

If you are going to write your own benchmarking utility -- use Benchmark module for heavy scripts and Time::HiRes module for very fast scripts (faster than 1 sec) where you need better time precision.

There is no need to write a special benchmark though. If you want to impress your boss or colleagues, just take some heavy CGI script you have (e.g. a script that crunches some data and prints the results to STDOUT), open 2 xterms and call the same script in mod_perl mode in one xterm and in mod_cgi mode in the other. You can use lwp-get from LWP package to emulate the web agent (browser). (benchmark directory of mod_perl distribution includes such an example)

2.21.1 Benchmarking scripts with execution times below 1 second :)

As noted before, for very fast scripts you will have to use the Time::HiRes module, its usage is similar to the Benchmark's.

```
use Time::HiRes qw(gettimeofday tv_interval);
my $start_time = [ gettimeofday ];
&sub_that_takes_a_teeny_bit_of_time()
my $end_time = [ gettimeofday ];
my $elapsed = tv_interval($start_time,$end_time);
print "the sub took $elapsed secs."
```

2.21.2 PerlHandler's Benchmarking

At <http://perl.apache.org/dist/contrib/> you will find Apache::Timeit package which does PerlHandler's Benchmarking.

2.22 Tuning the Apache's configuration variables for the best performance

It's very important to make a correct configuration of the MinSpareServers, MaxSpareServers, StartServers, MaxClients, and MaxRequestsPerChild parameters. There are no defaults, the values of these variable are very important, as if too "low" you will under-use the system's capabilities, and if too "high" chances that the server will bring the machine to its knees.

All the above parameters should be specified on the basis of the resources you have. While with a plain apache server, there is no big deal if you run too many servers (not too many of course) since the processes are of ~1Mb and aren't eating a lot of your RAM. Generally the numbers are even smaller if memory sharing is taking place. The situation is different with mod_perl. I have seen mod_perl processes

of 20Mb and more. Now if you have `MaxClients` set to 50: $50 \times 20\text{Mb} = 1\text{Gb}$ - do you have 1Gb of RAM? Probably not. So how do you tune these parameters? Generally by trying different combinations and benchmarking the server. Again `mod_perl` processes can be of much smaller size if sharing is in place.

Before you start this task you should be armed with a proper weapon. You need a **crashme** utility, which will load your server with `mod_perl` scripts you possess. You need it to have an ability to emulate a multiuser environment and to emulate multiple clients behavior which will call the `mod_perl` scripts at your server simultaneously. While there are commercial solutions, you can get away with free ones which do the same job. You can use an ApacheBench (`ab`) utility that comes with apache distribution, a `crashme` script which uses `LWP::Parallel::UserAgent` or `httpperf`.

Another important issue is to make sure to run testing client (load generator) on a system that is more powerful than the system being tested. After all we are trying to simulate the Internet users, where many users are trying to reach your service at once -- since a number of concurrent users can be quite large, your testing machine much be very powerful and capable to generate a heavy load. Of course you should not run the clients and the server on the same machine. If you do -- your testing results would be incorrect, since clients will eat a CPU and a memory that have to be dedicated to the server, and vice versa.

2.22.1 Tuning with *ab* - ApacheBench

ab is a tool for benchmarking your Apache HTTP server. It is designed to give you an impression on how much performance your current Apache installation can give. In particular, it shows you how many requests per secs your Apache server is capable of serving. The **ab** tool comes bundled with apache source distribution (and it's free :).

Let's try it. We will simulate 10 users concurrently requesting a very light script at `www.nowhere.com:81/test/test.pl`. Each "user" makes 10 requests.

```
% ./ab -n 100 -c 10 www.nowhere.com:81/test/test.pl
```

The results are:

```
Concurrency Level:      10
Time taken for tests:    0.715 seconds
Complete requests:      100
Failed requests:        0
Non-2xx responses:      100
Total transferred:      60700 bytes
HTML transferred:       31900 bytes
Requests per second:    139.86
Transfer rate:          84.90 kb/s received
```

```
Connection Times (ms)
      min      avg      max
Connect:    0       0       3
Processing: 13      67      71
Total:      13      67      74
```

The only numbers we really care about are:

```
Complete requests:    100
Failed requests:      0
Requests per second:  139.86
```

Let's raise the load of requests to 100 x 10 (10 users, each makes 100 requests)

```
% ./ab -n 1000 -c 10 www.nowhere.com:81/perl/access/access.cgi
Concurrency Level:    10
Complete requests:   1000
Failed requests:      0
Requests per second:  139.76
```

As expected nothing changes -- we have the same 10 concurrent users. Now let's raise the number of concurrent users to 50:

```
% ./ab -n 1000 -c 50 www.nowhere.com:81/perl/access/access.cgi
Complete requests:   1000
Failed requests:      0
Requests per second:  133.01
```

We see that the server is capable of serving 50 concurrent users at an amazing 133 req/sec! Let's find the upper boundary. Using `-n 10000 -c 1000` failed to get results (Broken Pipe?). Using `-n 10000 -c 500` derived 94.82 req/sec. The server's performance went down with the high load.

The above tests were performed with the following configuration:

```
MinSpareServers 8
MaxSpareServers 6
StartServers 10
MaxClients 50
MaxRequestsPerChild 1500
```

Now let's kill a child after a single request, we will use the following configuration:

```
MinSpareServers 8
MaxSpareServers 6
StartServers 10
MaxClients 100
MaxRequestsPerChild 1
```

Simulate 50 users each generating a total of 20 requests:

```
% ./ab -n 1000 -c 50 www.nowhere.com:81/perl/access/access.cgi
```

The benchmark timed out with the above configuration.... I watched the output of `ps` as I ran it, the parent process just wasn't capable of respawning the killed children at that rate...When I raised the `MaxRequestsPerChild` to 10 I've got 8.34 req/sec - very bad (18 times slower!) (You can't benchmark the importance of the `MinSpareServers`, `MaxSpareServers` and `StartServers` with this kind of test).

Now let's try to return `MaxRequestsPerChild` to 1500, but to lower the `MaxClients` to 10 and run the same test:

```
MinSpareServers 8
MaxSpareServers 6
StartServers 10
MaxClients 10
MaxRequestsPerChild 1500
```

I've got 27.12 req/sec, which is better but still 4-5 times slower (133 with `MaxClients` of 50)

Summary: I have tested a few combinations of server configuration variables (`MinSpareServers` `MaxSpareServers` `StartServers` `MaxClients` `MaxRequestsPerChild`). And the results we have received are as follows:

`MinSpareServers`, `MaxSpareServers` and `StartServers` are only important for user response times (sometimes user will have to wait a bit).

The important parameters are `MaxClients` and `MaxRequestsPerChild`. `MaxClients` should be not to big so it will not abuse your machine's memory resources and not too small, when users will be forced to wait for the children to become free to come serve them. `MaxRequestsPerChild` should be as big as possible, to take the full benefit of `mod_perl`, but watch your server at the beginning to make sure your scripts are not leaking memory, thereby causing your server (and your service) to die very fast.

Also it is important to understand that we didn't test the response times in the tests above, but the ability of the server to respond under a heavy load of requests. If the script that was used to test was heavier, the numbers would be different but the conclusions are very similar.

The benchmarks were run with:

```
HW: RS6000, 1Gb RAM
SW: AIX 4.1.5 . mod_perl 1.16, apache 1.3.3
Machine running only mysql, httpd docs and mod_perl servers.
Machine was _completely_ unloaded during the benchmarking.
```

After each server restart when I did changes to the server's configurations, I made sure the scripts were preloaded by fetching a script at least once by every child.

It is important to notice that none of requests timed out, even if was kept in server's queue for more than 1 minute! (That is the way **ab** works, which is OK for the testing purposes but will be unacceptable in the real world - users will not wait for more than 5-10 secs for a request to complete, and the client (browser) will timeout in a few minutes.)

Now let's take a look at some real code whose execution time is more than a few millisecs. We will do real testing and collect the data in tables for easier viewing.

I will use the following abbreviations:

```

NR    = Total Number of Request
NC    = Concurrency
MC    = MaxClients
MRPC  = MaxRequestsPerChild
RPS   = Requests per second

```

Running a mod_perl script with lots of mysql queries (the script under test is mysql bounded) (http://www.nowhere.com:81/perl/access/access.cgi?do_sub=query_form), with configuration:

```

MinSpareServers      8
MaxSpareServers     16
StartServers        10
MaxClients           50
MaxRequestsPerChild 5000

```

gives us:

NR	NC	RPS	comment
10	10	3.33	# not a reliable statistics
100	10	3.94	
1000	10	4.62	
1000	50	4.09	

Conclusions: Here I wanted to show that when the application is slow -- not due to perl loading, code compilation and execution, but bounded to some external operation like mysql querying which made the bottleneck -- it almost does not matter what load we place on the server. The RPS (Requests per second) is almost the same (given that all the requests have been served, you have an ability to queue the clients, but be aware that something that goes to queue means a waiting client and a client (browser) that might time out!)

Now we will benchmark the same script without using the mysql (perl only bounded code) (<http://www.nowhere.com:81/perl/access/access.cgi>), it's the same script that just returns a HTML form, without making any SQL queries.

```

MinSpareServers      8
MaxSpareServers     16
StartServers        10
MaxClients           50
MaxRequestsPerChild 5000

```

NR	NC	RPS	comment
10	10	26.95	# not a reliable statistics
100	10	30.88	
1000	10	29.31	
1000	50	28.01	
1000	100	29.74	
10000	200	24.92	
100000	400	24.95	

Conclusions: This time the script we executed was pure perl (not bounded to I/O or mysql), so we see that the server serves the requests much faster. You can see the `RequestPerSecond` (RPS) is almost the same for any load, but goes lower when the number of concurrent clients goes beyond the `MaxClients`. With 25 RPS, the client supplying a load of 400 concurrent clients will be served in 16 secs. But to get more realistic and assume the max concurrency of 100, with 30 RPS, the client will be served in 3.5 secs, which is pretty good for a highly loaded server.

Now we will use the server for its full capacity, by keeping all `MaxClients` alive all the time and having a big `MaxRequestsPerChild`, so no server will be killed during the benchmarking.

```
MinSpareServers      50
MaxSpareServers      50
StartServers         50
MaxClients           50
MaxRequestsPerChild 5000
```

NR	NC	RPS	comment
100	10	32.05	
1000	10	33.14	
1000	50	33.17	
1000	100	31.72	
10000	200	31.60	

Conclusion: In this scenario there is no overhead involving the parent server loading new children, all the servers are available, and the only bottleneck is contention for the CPU.

Now we will try to change the `MaxClients` and to watch the results: Let's reduce MC to 10.

```
MinSpareServers      8
MaxSpareServers      10
StartServers         10
MaxClients           10
MaxRequestsPerChild 5000
```

NR	NC	RPS	comment
10	10	23.87	# not a reliable statistics
100	10	32.64	
1000	10	32.82	
1000	50	30.43	
1000	100	25.68	
1000	500	26.95	
2000	500	32.53	

Conclusions: A very little difference! Almost no change! 10 servers were able to serve almost with the same throughput as 50 servers. Why? My guess it's because of CPU throttling. It seems that 10 servers were serving requests 5 times faster than when in the test above we worked with 50 servers. In the case above each child received its CPU time slice 5 times less frequently. So having a big value for `MaxClients`, doesn't mean that the performance will be better. You have just seen the numbers!

Now we will start to drastically reduce the `MaxRequestsPerChild`:

```
MinSpareServers      8
MaxSpareServers      16
StartServers         10
MaxClients           50
```

NR	NC	MRPC	RPS	comment
100	10	10	5.77	
100	10	5	3.32	
1000	50	20	8.92	
1000	50	10	5.47	
1000	50	5	2.83	
1000	100	10	6.51	

Conclusions: When we drastically reduce the `MaxRequestsPerChild`, the performance starts to become closer to the plain `mod_cgi`. Just for comparison with `mod_cgi`, here are the numbers of this run with `mod_cgi`:

```
MinSpareServers      8
MaxSpareServers      16
StartServers         10
MaxClients           50
```

NR	NC	RPS	comment
100	10	1.12	
1000	50	1.14	
1000	100	1.13	

Conclusion: `mod_cgi` is much slower :) in test `NReq/NClients 100/10` the RPS in `mod_cgi` was of 1.12 and in `mod_perl` of 32, which is 30 times faster!!! In the first test each child waited about 100 secs to be served. In the second and third 1000 secs!

2.22.2 Tuning with httpperf

`httpperf` is an utility written by David Mosberger. Just like `ApacheBench`--it measures the performance of the webserver.

A sample command line is shown below:

```
httpperf --server hostname --port 80 --uri /test.html \
--rate 150 --num-conn 27000 --num-call 1 --timeout 5
```

This command causes `httpperf` to use the web server on the host with IP name `hostname`, running at port 80. The web page being retrieved is `/test.html` and, in this simple test, the same page is retrieved repeatedly. The rate at which requests are issued is 150 per second. The test involves initiating a total of 27,000 TCP connections and on each connection one HTTP call is performed (a call consists of sending a request and receiving a reply).

The timeout option defines the number of seconds that the client is willing to wait to hear back from the server. If this timeout expires, the tool considers the corresponding call to have failed. Note that with a total of 27,000 connections and a rate of 150 per second, the total test duration will be approximately 180 seconds (27,000/150), independent of what load the server can actually sustain. And here is a result that one might get:

```
Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s

Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)
Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0
Connection time [ms]: connect 0.3

Request rate: 148.3 req/s (6.7 ms/req)
Request size [B]: 72.0

Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)
Reply time [ms]: response 4.6 transfer 0.0
Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)
Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0

CPUtime [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)
Net I/O: 190.9 KB/s (1.6*10^6 bps)

Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

2.22.3 Tuning with crashme script

This is another crashme suite originally written by Michael Schilli and located at <http://www.linux-magazin.de/ausgabe.1998.08/Pounder/pounder.html> . I did a few modifications (mostly adding my () operands). I also allowed it to accept more than one url to test, since sometimes you want to test an overall and not just one script.

The tool provides the same results as **ab** above but it also allows you to set the timeout value, so requests will fail if not served within the time out period. You also get **Latency** (secs/Request) and **Throughput** (Requests/sec) numbers. It can give you a better picture and make a complete simulation of your favorite Netscape browser :).

I have noticed while running these 2 benchmarking suites - **ab** gave me results 2.5-3.0 times better. Both suites run on the same machine with the same load with the same parameters. But the implementations are different.

Sample output:

```
URL(s):          http://www.nowhere.com:81/perl/access/access.cgi
Total Requests: 100
Parallel Agents: 10
Succeeded:       100 (100.00%)
Errors:          NONE
Total Time:      9.39 secs
Throughput:      10.65 Requests/sec
Latency:         0.85 secs/Request
```

And the code:

```
#!/usr/apps/bin/perl -w

use LWP::Parallel::UserAgent;
use Time::HiRes qw(gettimeofday tv_interval);
use strict;

###
# Configuration
###

my $nof_parallel_connections = 10;
my $nof_requests_total = 100;
my $timeout = 10;
my @urls = (
    'http://www.nowhere.com:81/perl/faq_manager/faq_manager.pl',
    'http://www.nowhere.com:81/perl/access/access.cgi',
);

#####
# Derived Class for latency timing
#####

package MyParallelAgent;
@MyParallelAgent::ISA = qw(LWP::Parallel::UserAgent);
use strict;

###
# Is called when connection is opened
###
sub on_connect {
    my ($self, $request, $response, $entry) = @_;
    $self->{__start_times}->{$entry} = [Time::HiRes::gettimeofday];
}

###
# Are called when connection is closed
###
sub on_return {
    my ($self, $request, $response, $entry) = @_;
    my $start = $self->{__start_times}->{$entry};
    $self->{__latency_total} += Time::HiRes::tv_interval($start);
}

sub on_failure {
    on_return(@_); # Same procedure
}

###
# Access function for new instance var
###
sub get_latency_total {
    return shift->{__latency_total};
}
```

```
#####
package main;
#####
###
# Init parallel user agent
###
my $ua = MyParallelAgent->new();
$ua->agent("pounder/1.0");
$ua->max_req($nof_parallel_connections);
$ua->redirect(0); # No redirects

###
# Register all requests
###
foreach (1..$nof_requests_total) {
  foreach my $url (@urls) {
    my $request = HTTP::Request->new('GET', $url);
    $ua->register($request);
  }
}

###
# Launch processes and check time
###
my $start_time = [gettimeofday];
my $results = $ua->wait($timeout);
my $total_time = tv_interval($start_time);

###
# Requests all done, check results
###

my $succeeded = 0;
my %errors = ();

foreach my $entry (values %$results) {
  my $response = $entry->response();
  if($response->is_success()) {
    $succeeded++; # Another satisfied customer
  } else {
    # Error, save the message
    $response->message("TIMEOUT") unless $response->code();
    $errors{$response->message}++;
  }
}

###
# Format errors if any from %errors
###
my $errors = join(',', map "$_ ($errors{$_})", keys %errors);
$errors = "NONE" unless $errors;

###
# Format results
###

#@urls = map {($_, ".")} @urls;
```


There is no problem -- any connection attempts over the `MaxClients` limit will normally be queued, up to a number based on the `ListenBacklog` directive. Once a child process is freed at the end of a different request, the connection will then be served.

But it **is an error** because clients are being put in the queue rather than getting served at once, despite the fact that they do not get an error response. The error can be allowed to persist to balance available system resources and response time, but sooner or later you will need to get more RAM so you can start more children. The best approach is to try not to have this condition reached at all, and if reach it often you should start to worry about it.

It's important to understand how much real memory a child occupies. Your children can share the memory between them (when OS supports that and you take action to allow the sharing happen. If this is the case, chances are that your `MaxClients` can be even higher. But it seems that it's not so simple to calculate the absolute number. (If you come up with solution please let us know!). If the shared memory was of the same size through the child's life, we could derive a much better formula:

$$\text{MaxClients} = \frac{\text{Total_RAM} + \text{Shared_RAM_per_Child} * \text{MaxClients}}{\text{Max_Process_Size} - 1}$$

which is:

$$\text{MaxClients} = \frac{\text{Total_RAM} - \text{Max_Process_Size}}{\text{Max_Process_Size} - \text{Shared_RAM_per_Child}}$$

Let's roll some calculations:

```
Total_RAM           = 500Mb
Max_Process_Size    = 10Mb
Shared_RAM_per_Child = 4Mb
```

$$\text{MaxClients} = \frac{500 - 10}{10 - 4} = 81$$

With no sharing in place

$$\text{MaxClients} = \frac{500}{10} = 50$$

With sharing in place you can have 60% more servers without purchasing more RAM, if you improve and keep the sharing level, let's say:

```
Total_RAM           = 500Mb
Max_Process_Size    = 10Mb
Shared_RAM_per_Child = 8Mb
```

$$\text{MaxClients} = \frac{500 - 10}{10 - 8} = 245$$

390% more servers!!! You've got the point :)

2.22.5 *Choosing MaxRequestsPerChild*

The `MaxRequestsPerChild` directive sets the limit on the number of requests that an individual child server process will handle. After `MaxRequestsPerChild` requests, the child process will die. If `MaxRequestsPerChild` is 0, then the process will live forever.

Setting `MaxRequestsPerChild` to a non-zero limit has two beneficial effects: it solves memory leak-ages and helps reduce the number of processes when the server load reduces.

The first reason is the most crucial for `mod_perl`, since sloppy programming will cause a child process to consume more memory after each request. If left unbounded, then after a certain number of requests the children will use up all the available memory and leave the server to die from memory starvation. Note, that sometimes standard system libraries leak memory too, especially on OSes with bad memory management (e.g. Solaris 2.5 on x86 arch). If this is your case you can set `MaxRequestsPerChild` to a small number, which will allow the system to reclaim the memory, greedy child process consumed, when it exits after `MaxRequestsPerChild` requests. But beware -- if you set this number too low, you will lose a fraction of the speed bonus you receive with `mod_perl`. Consider using `Apache::PerlRun` if this is the case. Also setting `MaxSpareServers` to a number close to `MaxClients`, will improve the response time (but your parent process will be busy respawning new children all the time!)

Another approach is to use `Apache::SizeLimit`. By using this module, you should be able to discontinue using the `MaxRequestsPerChild`, although for some folks, using both in combination does the job.

2.22.6 *Choosing MinSpareServers, MaxSpareServers and StartServers*

With `mod_perl` enabled, it might take as much as 30 seconds from the time you start the server until it is ready to serve incoming requests. This delay depends on the OS, the number of preloaded modules and the process load of the machine. So it's best to set `StartServers` and `MinSpareServers` to high numbers, so that if you get a high load just after the server has been restarted, the fresh servers will be ready to serve requests immediately. With `mod_perl`, it's usually a good idea to raise all 3 variables higher than normal. In order to maximize the benefits of `mod_perl`, you don't want to kill servers when they are idle, rather you want them to stay up and available to immediately handle new requests. I think an ideal configuration is to set `MinSpareServers` and `MaxSpareServers` to similar values, maybe even the same. Having the `MaxSpareServers` close to `MaxClients` will completely use all of your resources (if `MaxClients` has been chosen to take the full advantage of the resources), but it'll make sure that at any given moment your system will be capable of responding to requests with the maximum speed (given that number of concurrent requests is not higher than `MaxClients`.)

Let's try some numbers. For a heavily loaded web site and a dedicated machine I would think of (note 400Mb is just for example):

```

Available to webserver RAM: 400Mb
Child's memory size bounded: 10Mb
MaxClients: 400/10 = 40 (larger with mem sharing)
StartServers: 20
MinSpareServers: 20
MaxSpareServers: 35

```

However if I want to use the server for many other tasks, but make it capable of handling a high load, I'd think of:

```

Available to webserver RAM: 400Mb
Child's memory size bounded: 10Mb
MaxClients: 400/10 = 40
StartServers: 5
MinSpareServers: 5
MaxSpareServers: 10

```

(These numbers are taken off the top of my head, and it shouldn't be used as a rule, but rather as examples to show you some possible scenarios. Use this information wisely!)

2.22.7 Summary of Benchmarking to tune all 5 parameters

OK, we've run various benchmarks -- let's summarize the conclusions:

- **MaxRequestsPerChild**

If your scripts are clean and don't leak memory, set this variable to a number as large as possible (10000?). If you use `Apache::SizeLimit`, you can set this parameter to 0 (equal to infinity). You will want this parameter to be smaller if your code becomes unshared over the process' life.

- **StartServers**

If you keep a small number of servers active most of the time, keep this number low. Especially if `MaxSpareServers` is low as it'll kill the just loaded servers before they were utilized at all (if there is no load). If your service is heavily loaded, make this number close to `MaxClients` (and keep `MaxSpareServers` equal to `MaxClients` as well.)

- **MinSpareServers**

If your server performs other work besides web serving, make this low so the memory of unused children will be freed when there is no big load. If your server's load varies (you get loads in bursts) and you want fast response for all clients at any time, you will want to make it high, so that new children will be respawned in advance and be waiting to handle bursts of requests.

- **MaxSpareServers**

The logic is the same as of `MinSpareServers` - low if you need the machine for other tasks, high if it's a dedicated web host and you want a minimal response delay.

- **MaxClients**

Not too low, so you don't get into a situation where clients are waiting for the server to start serving them (they might wait, but not for too long). Do not set it too high, since if you get a high load and all requests will be immediately granted and served, your CPU will have a hard time keeping up, and if the child's size * number of running children is larger than the total available RAM, your server will start swapping (which will slow down everything, which in turn will make things even more slower, until eventually your machine will die). It's important that you take pains to ensure that swapping does not normally happen. Swap space is an emergency pool, not a resource to be used on a consistent basis. If you are low on memory and you badly need it - buy it, memory is amazingly cheap these days.

But based on the test I conducted above, even if you have plenty of memory like I have (1Gb), increasing `MaxClients` sometimes will give you no speedup. The more clients are running, the more CPU time will be required, the less CPU time slices each process will receive. The response latency (the time to respond to a request) will grow, so you won't see the expected improvement. The best approach is to find the minimum requirement for your kind of service and the maximum capability of your machine. Then start at the minimum and test like I did, successively raising this parameter until you find the point on the curve of the graph of the latency or/and throughput where the improvement becomes smaller. Stop there and use it. Of course when you use these parameters in production server, you will have the ability to tune them more precisely, since then you will see the real numbers. Also don't forget that if you add more scripts, or just modify the running ones -- most probably that the parameters need to be recalculated, since the processes will grow in size as you compile in more code.

2.23 Persistent DB Connections

Another popular use of `mod_perl` is to take advantage of its ability to maintain persistent open database connections. The basic approach is as follows:

```
# Apache::Registry script
-----
use strict;
use vars qw($dbh);

$dbh ||= SomeDbPackage->connect(...);
```

Since `$dbh` is a global variable for the child, once the child has opened the connection it will use it over and over again, unless you perform `disconnect()`.

Be careful to use different names for handlers if you open connection to different databases!

`Apache::DBI` allows you to make a persistent database connection. With this module enabled, every `connect()` request to the plain DBI module will be forwarded to the `Apache::DBI` module. This looks to see whether a database handle from a previous `connect()` request has already been opened, and if this handle is still valid using the ping method. If these two conditions are fulfilled it just returns the database handle. If there is no appropriate database handle or if the ping method fails, a new connection is established and the handle is stored for later re-use. **There is no need to delete the `disconnect()`**

statements from your code. They will not do a thing, as the `Apache::DBI` module overloads the `disconnect()` method with a NOP. On child's exit there is no explicit disconnect, the child dies and so does the database connection. You may leave the `use DBI;` statement inside the scripts as well.

The usage is simple -- add to `httpd.conf`:

```
PerlModule Apache::DBI
```

It is important, to load this module before any other `DBI`, `DBD::*` and `ApacheDBI*` modules!

```
db.pl
-----
use DBI;
use strict;

my $dbh = DBI->connect( 'DBI:mysql:database', 'user', 'password',
                      { autocommit => 0 }
                      ) || die $DBI::errstr;

...rest of the program
```

2.23.1 Preopening Connections at the Child Process' Fork Time

If you use `DBI` for DB connections, and you use `Apache::DBI` to make them persistent, it also allows you to preopen connections to DB for each child with `connect_on_init()` method, thus saving up a connection overhead on the very first request of every child.

```
use Apache::DBI ();
Apache::DBI->connect_on_init("DBI:mysql:test",
                             "login",
                             "passwd",
                             {
                               RaiseError => 1,
                               PrintError => 0,
                               AutoCommit => 1,
                             }
                             );
```

This can be used as a simple way to have apache children establish connections on server startup. This call should be in a startup file `require()`d by `PerlRequire` or inside `<Perl>` section. It will establish a connection when a child is started in that child process. See the `Apache::DBI` manpage to see the requirements for this method.

2.23.2 Caching `prepare()` statements

You can also benefit from persistent connections by replacing `prepare()` with `prepare_cached()`. That way you will always be sure that you have a good statement handle and you will get some caching benefit. The downside is that you are going to pay for `DBI` to parse your SQL and do a cache lookup every time you call `prepare_cached()`.

Be warned that some databases doesn't support caches of prepared plans. (e.g PostgreSQL and Sybase). Though with Sybase you could open multiple connections to achieve the same result (at the risk of getting deadlocks depending on what you are trying to do!)

2.23.3 Handling Timeouts

Another problem is with timeouts: some databases disconnect the client after a certain time of inactivity. This problem is known as **morning bug**. The `ping()` method ensures that this will not happen. Some DBD drivers don't have this method, check the `Apache::DBI` manpage to see how to write a `ping()` method.

Another approach is to change the client's connection timeout. For mysql users, starting from mysql-3.22.x you can set a `wait_timeout` option at mysql server startup to change the default value. Setting it to 36 hours probably would fix the timeout problem.

2.24 Using \$|=1 under mod_perl and better print() techniques.

As you know `local $|=1;` disables the buffering of the currently selected file handle (default is STDOUT). If you enable it, `ap_rflush()` is called after each `print()`, unbuffering Apache's IO.

If you are using a `_bad_` style in generating output, which consist of multiple `print()` calls, or you just have too many of them, you will experience a degradation in performance. The severity depends on the number of the calls you make.

Many old CGIs were written in the style of:

```
print "<BODY BGCOLOR=\"black\" TEXT=\"white\">";
print "<H1>";
print "Hello";
print "</H1>";
print "<A HREF=\"foo.html\"> foo </A>";
print "</BODY>";
```

which reveals the following drawbacks: multiple `print()` calls - performance degradation with `$|=1`, backslashism which makes the code less readable and more difficult to format the HTML to be easily readable as CGI's output. The code below solves them all:

```
print qq{
  <BODY BGCOLOR="black" TEXT="white">
    <H1>
      Hello
    </H1>
    <A HREF="foo.html"> foo </A>
  </BODY>
};
```

I guess you see the difference. Be careful though, when printing a <HTML> tag. The correct way is:

```
print qq{<HTML>
  <HEAD></HEAD>
  <BODY>
}
```

If you try the following:

```
print qq{
  <HTML>
  <HEAD></HEAD>
  <BODY>
}
```

Some older browsers might not accept the output as HTML, but rather print it as a plain text, since they expect the first characters after the headers and empty line to be <HTML> and not spaces and/or additional newline and then <HTML>. Even if it works with your browser, it might not work for others.

Now let's go back to the \$|=1 topic. I still disable buffering, for 2 reasons: I use few `print()` calls by printing out multiline HTML and not a line per `print()` and I want my users to see the output immediately. So if I am about to produce the results of the DB query, which might take some time to complete, I want users to get some titles ahead. This improves the usability of my site. Recall yourself: What do you like better: getting the output a bit slower, but steadily from the moment you've pressed the **Submit** button or having to watch the "falling stars" for awhile and then to receive the whole output at once, even a few millisecs faster (if the client (browser) did not time out till then).

An even better solution is to keep the buffering enabled, and use a Perl API `rflush()` call to flush the buffers when wanted. This way you can aggregate in the buffer the top of the page you are going to send to user, and flush it a moment before you are going to do some lengthy operation, like DB query. So you kill the two birds in one shoot: You show some of the data to the user immediately, so user will feel that something is actually happening, and you almost have no performance hit caused by disabled buffering.

```
use CGI ();
my $r = shift;
my $q = new CGI;
print $q->header('text/html');
print $q->start_html;
print $q->p("Searching...Please wait");
$r->rflush;
# imitate a lengthy operation
for (1..5) {
  sleep 1;
}
print $q->p("Done!");
```

Conclusion: Do not blindly follow suggestions, but think what is best for you in every given case.

2.25 Avoid Importing Functions

When possible, avoid importing a module's functions into your name space. The aliases which are created can take up quite a bit of space. Try to use method interfaces and fully qualified `Package::function` or `$Package::variable` like names instead.

2.26 Object Methods Calls Versus Function Calls

Which subroutine calling form is more efficient: OOP methods or functions?

2.26.1 *The Overhead with Light Subroutines*

Let's do a benchmarking. We will start doing it using empty methods, which will allow us to measure the real difference in the overhead each kind of call introduces. We will use this code:

```
bench_call1.pl
-----
package Foo;

use strict;
use Benchmark;

sub bar { };

timethese(50_000, {
    method => sub { Foo->bar() },
    function => sub { Foo::bar('Foo') },
});
```

The two calls are equivalent, since both pass the class name as their first name, *function* does this explicitly, while *method* does this transparently.

The benchmarking result:

```
Benchmark: timing 50000 iterations of function, method...
function:  0 wallclock secs ( 0.80 usr + 0.05 sys = 0.85 CPU)
method:   1 wallclock secs ( 1.51 usr + 0.08 sys = 1.59 CPU)
```

We are interested in the 'total CPU times' and not the 'wallclock seconds'. It's possible that the load on the system was different for the two tests while benchmarking, so the wallclock times give us no useful information.

We see that the *method* calling type is almost twice slower, than *function*. 0.85 CPU compared to 1.59 CPU real execution time. Why does this happen? Because the difference between functions and methods is the time taken to resolve the pointer from the object, to find the module it belongs to and then the actual method. Functions form has one parameter less to pass, less stack operations, less time to get to the guts of the subroutine.

2.26.2 The Overhead with Heavy Subroutines

But that doesn't mean that you shouldn't use methods. Generally your functions do something, and the more they do the less will be the difference, because the overhead time is a number of a final length. Therefore the longer execution time of the function the smaller the relative overhead of the method call. The next benchmark proves this point:

```
bench_call2.pl
-----
package Foo;

use strict;
use Benchmark;

sub bar {
    my $class = shift;

    my ($x,$y) = (100,100);
    $y = log ($x ** 10) for (0..20);
};

timethese(50_000, {
    method => sub { Foo->bar() },
    function => sub { Foo::bar('Foo'); },
});
```

We get a very close benchmarks!

```
function: 33 wallclock secs (15.81 usr + 1.12 sys = 16.93 CPU)
method: 32 wallclock secs (18.02 usr + 1.34 sys = 19.36 CPU)
```

Let's make the subroutine *bar* even slower:

```
sub bar {
    my $class = shift;

    my ($x,$y) = (100,100);
    $y = log ($x ** 10) for (0..40);
};
```

And the result is amazing, the *method* call convention was faster then *function*:

```
function: 81 wallclock secs (25.63 usr + 1.84 sys = 27.47 CPU)
method: 61 wallclock secs (19.69 usr + 1.49 sys = 21.18 CPU)
```

In case your functions do very little, like the functions that generate HTML tags in CGI.pm, the overhead might become a significant one. If your goal is speed you might consider to use the *function* form, but if you write a big and complicated application, it's much better to use the *method* form, as it will make your code easier to develop, maintain and debug.

2.26.3 Are All Methods Slower than Functions?

Some modules' API is misleading, for example `CGI.pm` allows you to execute its subroutines as functions and methods. As you will see in a moment its function form of the calls is slower than the method form because it does some woodoo work when the function form call is used.

```
use CGI;
my $q = new CGI;
$q->param('x',5);
my $x = $q->param('x');
```

versus

```
use CGI qw(:standard);
param('x',5);
my $x = param('x');
```

As usual, let's benchmark some very light calls and compare. Ideally we would expect the *methods* to be slower than *functions* based on the previous benchmarks:

```
bench_call13.pl
-----
use Benchmark;

use CGI qw(:standard);
$CGI::NO_DEBUG = 1;
my $q = new CGI;
my $x;
timethese
(20000, {
  method => sub {$q->param('x',5); $x = $q->param('x'); },
  function => sub { param('x',5); $x = param('x'); },
});
```

The benchmark is written in such a way that all the initializations are done at the beginning, so that we get as accurate performance figures as possible. Let's do it:

```
% ./bench_call13.pl

function: 51 wallclock secs (28.16 usr + 2.58 sys = 30.74 CPU)
method: 39 wallclock secs (21.88 usr + 1.74 sys = 23.62 CPU)
```

As we can see methods are faster than functions, which seems to be wrong. The explanation lies in the way `CGI.pm` is implemented. `CGI.pm` uses some *fancy* tricks to make the same routine act both as a *method* and a plain *function*. The overhead of checking whether the arguments list looks like a *method* invocation or not will mask the slight difference in time for the way the function was called.

If you are intrigued and want to investigate further by yourself the subroutine you want to explore is called *self_or_default*. The first line of this function short-circuits if you are using the object methods, but the whole function is called if you are using the functional forms. Therefore, the functional form should be slightly slower than the object form.

2.26.4 Imported Symbols and Memory Usage

There is a real memory hit when you import all of the function into your process' memory. This can significantly enlarge memory requirements, particularly when there are many child processes.

In addition to polluting the namespace, when a process imports symbols from any module or any script it grows by the size of the space allocated for those symbols. The more you import (e.g. `qw(:standard)` vs `qw(:all)`) the more memory will be used. Let's say the overhead is of size X. Now take the number of scripts in which you deploy the function method interface, let's call that Y. Finally let's say that you have a number of processes equal to Z.

You will need $X*Y*Z$ size of additional memory, taking $X=10k$, $Y=10$, $Z=30$, we get $10k*10*30 = 3Mb!!!$ Now you understand the difference.

Let's benchmark `CGI.pm` using `GTop.pm`. First we will try it with no exporting at all.

```
use GTop ();
use CGI ();
print GTop->new->proc_mem($$)->size;

1,949,696
```

Now exporting a few dozens symbols:

```
use GTop ();
use CGI qw(:standard);
print GTop->new->proc_mem($$)->size;

1,966,080
```

And finally exporting all the symbols (about 130)

```
use GTop ();
use CGI qw(:all);
print GTop->new->proc_mem($$)->size;

1,970,176
```

Results:

import symbols	size(bytes)	delta(bytes)	relative to ()
()	1949696	0	
qw(:standard)	1966080	16384	
qw(:all)	1970176	20480	

So in my example above $X=20k \Rightarrow 20K*10*30 = 6Mb$. You will need 6Mb more when importing all the `CGI.pm`'s symbols than when you import none at all.

Generally you use more than one script, run more than one process and probably import more symbols from the additional modules that you deploy. So the real numbers are much bigger.

The function method is faster in the general case, because of the time overhead to resolve the pointer from the object.

If you are looking for performance improvement, you will have to face the fact that having to type `My::Module::my_method` might save you a good chunk of memory if the above call must not be called with a reference to an object, but even then it can be passed by value.

I strongly endorse *Apache::Request (libapreq) - Generic Apache Request Library*. Its is written in C, giving it a significant memory and performance benefit. It has all the functionality of `CGI.pm` except HTML generation functions.

2.27 Sending plain HTML as a compressed output

Have you ever served a huge HTML file (e.g. a file bloated with JavaScript code) and wondered how could you send it compressed, thus dramatically cutting down the download times. After all java applets can be compressed into a jar and benefit from a faster download times. Why cannot we do the same with a plain ASCII (HTML,JS and etc), it is a known fact that ASCII text can be compressed by a factor of 10.

`Apache::GzipChain` comes to help you with this task. If a client (browser) understands `gzip` encoding this module compresses the output and sends it downstream. A client decompresses the data upon receive and renders the HTML as if it was a plain HTML fetch.

For example to compress all html files on the fly, do:

```
<Files *.html>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::PassFile
</Files>
```

Remember that it will work only if the browser claims to accept compressed input, thru `Accept-Encoding` header. `Apache::GzipChain` keeps a list of user-agents, thus it also looks at `User-Agent` header, for known to accept compressed output browsers.

For example if you want to return compressed files which should pass in addition through `Embperl` module, you would write:

```
<Location /test>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::EmbperlChain Apache::PassFile
</Location>
```

Hint: Watch an `access_log` file to see how many bytes were actually send, compare with a regular configuration send.

(See `perldoc Apache::GzipChain`).

Notice that the rightmost PerlHandler must be a content producer. Use `Apache::PassFile` or another similar module.

;o)

3 Choosing an Operating System and Hardware

3.1 What we will learn in this chapter

- Is it important?
- Choosing an Operating System
- Choosing Hardware

3.2 Is it important?

You can invest a lot of time and money into server tuning and code rewriting according the guidelines you have just learned, but your performance will be really bad if you do not take into account the hardware demands, and do not wisely choose the operating system suited for your needs. While the tips below apply to any webserver, they are written for an administrator of a mod_perl-enabled webserver

3.3 Choosing an Operating System

First let's talk about Operating Systems (OS). While I am personally a Linux devotee, I do not want to start yet another OS war. Assuming this, I will try to define what you should be looking for, then when you know what do you want from your OS, go find it. Visit the Web sites of operating systems you are interested in. You can gauge user's opinions by searching relevant discussions in newsgroup and mailing list archives such as Deja - <http://deja.com> and eGroups - <http://egroups.com> . I will leave this fan research up to you. But I would use Linux or something from the *BSD family.

3.3.1 *Stability and Robustness*

Probably the most desired features in an OS are stability and robustness. You are in an Internet business, which does not have normal working hours, like many conventional businesses you know about (9am to 5pm). You are open 24 hours a day. You cannot afford to be off-line, for your customers will go shop at another service like yours, unless you have a monopoly :) . If the OS of your choice crashes every day or so, I would throw it away, after doing a little investigation, for there might be a reason for a system crash. Like a runaway server that eats up all the memory and disk, so you cannot blame the OS for that. Generally, people who use the OS for some time can tell you a lot about its stability.

3.3.2 *Memory Management*

You want an OS with a good memory management, some OSes are well known as memory hogs. The same code can use twice as much memory on one OS compared to the other. If the size of the mod_perl process is 10Mb and you have tens of these running, it definitely adds up!

3.3.3 Memory Leakages

Some OSes and/or the libraries (like C runtime libraries) suffer from memory leaks. You cannot afford such a system, for you already know that a single mod_perl process sometimes serves thousands of requests before it terminates. So if a leak occurs on every request, your memory demands will be huge. Of course your code can be the cause of the memory leaks as well (check out the `Apache::Leak` module). Certainly, you can lower the number of requests to be served over the process' life, but that can degrade performance.

3.3.4 Sharing Memory

You want an OS with good memory sharing capabilities. As you have learned, if you preload the modules and scripts at server startup, they are shared between the spawned children, at least for a part of a process' life span, since memory pages become "dirty" and cease to be shared. This feature can save you up a lot of memory!

3.3.5 Cost and Support

If you are in a big business you probably do not mind paying another \$1000 for some fancy OS and to get the bundled support for it. But if your resources are low, you will look for cheaper and free OS. Free does not mean bad, it can be quite opposite as we all either know from our own experience or read about in news. Free OSes could have and do have the best support you can find. It is very easy to understand - most of the people are not rich and will try to use a cheaper or free OS first if it does the work for them. Since it really fits their needs, many people keep using it and eventually know it well enough to be able to provide support for others in trouble. Why would they do this for free? For the spirit of the first days of the Internet, when there was no commercial Internet and people helped each other, because someone helped them in first place. I was there, I was touched by that spirit and I will do anything to keep that spirit alive.

But, let's get back to our world. We are living in material world, and our bosses pay us to keep the systems running. So if you feel that you cannot provide the support yourself and you do not trust the available free resources, you must pay for an OS backed by a company, and blame them for any problem. Your boss wants to be able to sue someone if the project has a problem caused by the external product that is being used in the project. If you buy a product and the company selling it, claims support, you have someone to sue. You do not have someone to sue other than getting yourself fired if you go with Open Source and it fails.

Also remember that if you spend less or zero money on OS and Software, you will be able to buy a better and stronger hardware.

3.3.6 Discontinued products

You have invested a lot of time and money into developing some proprietary software that is bundled with the OS you were developing on. Like writing a mod_perl handler that takes advantage of some proprietary features of the OS and it will not run on any other OS. Things are under control, the performance is great and you sing from happiness. But... one day the company who wrote your beloved OS goes bankrupt,

which is not unlikely to happen nowadays. You are stuck with their last masterpiece and no support! What you are going to do then? Invest more into porting the software to another OS...

Everyone can be hit by this mini-disaster, so it is better to check the background of the company when making your choice, but still you never know what will happen tomorrow. The OSes in this hazard group are completely developed by a single companies. Free OSes are probably less susceptible to this, for development is distributed between many companies and developers, so if a person who developed a really important part of the kernel lost interest in continuing, someone else will pick the falling flag and carry on. Of course if tomorrow some better project showed up, developers might migrate there and finally drop the development, but we are here not to let this happen.

In the final analysis, the decision is yours.

3.3.7 OS Releases

Actively developed OSes generally try to keep the pace with the latest technology developments, and continually optimize the kernel and other parts of the OS to become better and faster. Nowadays, Internet and networking in general are the hottest targets for system developers. Sometimes a simple OS upgrade to a latest stable version, can save you an expensive hardware upgrade. Also, remember that when you buy new hardware, chances are that the latest software will make the most of it. Since the existing software (drivers) might support the brand new product because of its backwards compatibility with previous products of the same family, it might not reap all the benefits of the new features. It means that you could spend much less money for almost the same functionality if you were to buy a previous model of the same product.

3.4 Choosing Hardware

Since I am not fond of the idea of updating this section every day a new processor or memory type comes out, I will only hint what should you look for and suggest that sometimes the most expensive machine is not the one which provides the best performance.

Your demands are based on many aspects and components. Let's discuss some of them.

In discussion course you might meet some unfamiliar terms, here are some of them:

- Clustering - a bunch of machines connected together to perform one big or many small computational tasks in a reasonable time.
- Load balancing - users can remember only a name of one of your machines - namely of your server, but it cannot stand the heavy load, so you use a clustering approach, distributing the load over a number of machines. The central server, the one users access when they type the name of the service, works as a dispatcher, by redirecting requests to the rest of the machines, sometimes it also collects the results and return them to the users. One of the advantages is that you can take one of the machines down for a repair or upgrade, and your service will still work - the main server will not dispatch the requests to the machine that was taken down. I will just say that there are many load balancing techniques.

- NIC - Network Interface Card.
- RAM - Random Access Memory

3.4.1 Expected site traffic

If you are building a fan site, but want to amaze your friends with a mod_perl guest book, an old 486 machine will do it. If you are into a serious business, it is very important to build a scalable server, so if your service is successful and becomes popular, you get your server's traffic doubled every few days, you should be ready to add more resources dynamically. While we can define the webserver scalability more precisely, the important thing is to make sure that you can add more power to your `webserver(s)` without investing additional money into a software developing (almost, you will need a software to connect your servers if you add more of them). It means that you should choose a hardware/OS that can talk to other machines and become a part of the cluster.

From the other hand if you prepare for a big traffic and buy a monster to do the work for you, what happens if your service does not prove to be as successful as you thought it would be. Then you spent too much money and meanwhile there were a new faster processors and other hardware components released, so you loose again.

Wisdom and prophecy , that's all it takes :)

3.4.2 Cash

Everybody knows that Internet is a cash hole, what you throw in, hardly comes back. This is not always true, but there is a lot of wisdom in these words. While you have to invest money to build a decent service, it can be cheaper! You can spend as much as 10 times more money on a strong new machine, but get only a 10% improvement in performance. Remember that a four year old processor is still very powerful.

If you really need a lot of power do not think about a single strong machine (unless you have money to throw away), think about clustering and load balancing. You can probably buy 10 times more older but very cheap machines and have a 8 times more power, then purchasing only one single new machine. Why is that? Because as I mentioned before generally the performance improvement is marginal while the price is much bigger. Because 10 machines will do faster disk I/O, than one single machine, even if the disk is much faster. Yes, you have more administration overhead, but there is a chance you will have it anyway, for in a short time the machine you have just invested in will not stand the load anyway and you will have to purchase more and think how to implement load balancing and file system distribution.

Why I am so convinced? Facts! Look at the most used services on the Internet: search engines, email servers and the like -- most of them are using a clustering approach. While you may not always notice that, they do it by hiding the real implementation behind the proxy servers.

3.4.3 Internet Connection

You have the best hardware you can get, but the service is still crawling. Make sure you have a fast Internet connection. Not as fast as your ISP claims it to be, but fast as it should be. The ISP might have a very good connection to the Internet, but puts many clients on the same line. If these are heavy clients, your traffic will have to share the same line and the throughput will decline. Think about a dedicated connection and make sure it is truly dedicated. Trust the ISP but check it!

The idea of having a connection to **The Internet** is a little misleading. Many Web hosting and co-location companies have large amounts of bandwidth, but still have poor connectivity. The public exchanges, such as MAE-East and MAE-West, frequently become overloaded, yet many ISPs depend on these exchanges.

Private peering means that providers can exchange traffic much quicker.

Also, if your Web site is of global interest, check that the ISP has good global connectivity. If the Web site is going to be visited mostly by people in a certain country or region, your server should probably be located there.

And a bad connectivity can directly influence your machine's performance. Here is a story, one of the developers told on the `mod_perl` mailing list:

```
What relationship has 10% packet loss on one upstream provider got  
to do with machine memory ?
```

```
Yes.. a lot. For a nightmare week, the box was located downstream of  
a provider who was struggling with some serious bandwidth problems  
of his own... people were connecting to the site via this link, and  
packet loss was such that retransmits and tcp stalls were keeping  
httpd heavies around for much longer than normal.. instead of  
blasting out the data at high or even modem speeds, they would be  
stuck at 1k/sec or stalled out... people would press stop and  
refresh, httpds would take 300 seconds to timeout on writes to  
no-one.. it was a nightmare. Those problems didn't go away till I  
moved the box to a place closer to some decent backbones.
```

```
Note that with a proxy, this only keeps a lightweight httpd tied up,  
assuming the page is small enough to fit in the buffers. If you are  
a busy internet site you always have some slow clients. This is a  
difficult thing to simulate in benchmark testing, though.
```

3.4.4 I/O performance

If your service is I/O bound (does a lot of read/write operations to disk, remember that relational databases are sitting on disk as well) you need a very fast disk. So you should not spend money on Video card and monitor (monochrome card and 14" B&W are perfectly adequate for a server -- you will probably be telnetted or ssh-ed in most of the time), but rather look for disks with the best price/performance ratio. Of course, ask around and avoid disks that have a reputation for headcrashes and other disasters.

With money in hand you should think about getting a RAID system. RAID is generally a box with many HDs. It is capable of reading and writing data much faster, and is protected against disk failures. It does this by duplicating the same data over a number of disks, so if one fails, the RAID controller detects it and the data is still correct on the duplicated disks. You must think about RAID or similar systems if you have an enormous data set to serve. (What is an enormous data set nowadays? Gigabytes, terabytes?).

Ok, we have a fast disk, what's next? You need a fast disk controller. So either you should use the one embedded on your motherboard or you should plug a controller card if the one you have onboard is not good enough.

3.4.5 Memory

How much RAM (Randomly Accessed Memory) do you need? Nowadays, chances are you will hear: "Memory is cheap, the more you buy the better". But how much is enough? The answer pretty straightforward: "You do not want your machine to swap". When the CPU needs to write something into memory, but notices that it is already full, it takes the least frequently used memory pages and swaps them out. Swapping out means writing the data to disk. Another process then references some of its own data, which happens to be on one of the pages that were just swapped out. The CPU, ever obliging, swaps it back in again, probably swapping out some other data that will be needed very shortly by another process. Carried to the extreme, the CPU and disk start to thrash hopelessly in circles, without getting any real work done. The less RAM there is, the more often this scenario arises. Worse, you can exhaust swap space as well, and then the troubles really set in...

How do you make a decision? You know the highest rate your server expects to serve pages and how long it takes to do so. Now you can calculate how many server processes you need. Knowing the maximum size any of your servers can get, you know how much memory you need. You probably need less memory than you have calculated if your OS supports memory sharing and you know how to make best use of this feature (preloading the modules and scripts at server startup). Do not forget that other essential system processes need memory as well, so you should plan not only for the web server, but also take into account the other players. Remember that requests can be queued, so you can afford to let your client wait for a few moments until a server is available to serve it, your numbers will be more correct, since you generally do not have the highest load, but you should be ready to bear the peaks. So you need to reserve at least 20% of free memory for peak situations. Many sites have crashed a few moments after a big scoop about them was posted and unexpected number of requests suddenly came in. (This is called a Slashdot effect, which was born at <http://slashdot.org>) If you are about to announce something cool, be aware of the possible consequences.

3.4.6 Bottlenecks

The most important thing to understand is that you might use the most expensive components, but still get bad performance. Why? Let me introduce an annoying word: A bottleneck.

A machine is an aggregate of many big and small components. Each one of them may be a bottleneck. If you have a fast processor but a small amount of RAM (memory), the processor will be under-utilized waiting for the kernel to swap the memory pages in and out, because memory is too small to hold the most used ones. If you have a lot of memory and a fast processor and a fast disk, but a slow controller - the

performance will be bad, and you have wasted money.

Use a fast NIC (Network Interface Card) that does not create a bottleneck. If it is slow, the whole service is slow. This is the most important component, since webservers are much more network-bound than disk-bound!

3.4.7 Conclusion

To use your money optimally you have to understand the hardware very well, so you will know what to pick. Otherwise, you should hire a knowledgeable hardware consultants and employ him/her on a regular basis, since your demands will probably change as time goes by and your hardware will likewise be forced to adapt as well.

;o)

4 Getting Help and Further Learning

4.1 What we will learn in this chapter

- Getting help
- Get help with mod_perl
- Get help with Perl
- Get help with Perl/CGI
- Get help with Apache
- Get help with DBI
- Get help with Squid

4.2 Getting help

If after reading this guide and other documents listed in this section, you feel that your question is not yet answered, please ask the apache/mod_perl mailing list to help you. But first try to browse the mailing list archive. Most of the time you will find the answer for your question by searching the mailing archive, since there is a big chance someone else has already encountered the same problem and found a solution for it. If you ignore this advice, do not be surprised if your question will be left unanswered - it bores people to answer the same question more than once. It does not mean that you should avoid asking questions. Just do not abuse the available help and **RTFM** before you call for **HELP**. (You have certainly heard the infamous fable of the shepherd boy and the wolves)

4.3 Get help with mod_perl

- **mod_perl home**
<http://perl.apache.org>
- **mod_perl Garden project**
<http://modperl.sourcegarden.org>
- **mod_perl Books**
 - **'Apache Modules' Book**

<http://www.modperl.com> is the home site of The Apache Modules Book, a book about creating Web server modules using the Apache API, written by Lincoln Stein and Doug MacEachern.

Now you can purchase the book at your local bookstore or from the online dealer. O'Reilly lists this book as:

Writing Apache Modules with Perl and C
By Lincoln Stein & Doug MacEachern
1st Edition March 1999
1-56592-567-X, Order Number: 567X
746 pages, \$34.95

○ **'Enabling web services with mod_perl' Book**

<http://www.modperlbook.com> is the home site of the new mod_perl book, that Eric Cholet and Stas Bekman are co-authoring together. We expect the book to be published in fall 2000.

Ideas, suggestions and comments are welcome. You may send them to info@modperlbook.com

● **mod_perl Guide**

by Stas Bekman at <http://perl.apache.org/guide>

● **mod_perl FAQ**

by Frank Cringle at <http://perl.apache.org/faq/> .

● **mod_perl performance tuning guide**

by Vivek Khera at <http://perl.apache.org/tuning/> .

● **mod_perl plugin reference guide**

by Doug MacEachern at http://perl.apache.org/src/mod_perl.html .

● **Quick guide for moving from CGI to mod_perl**

at http://perl.apache.org/dist/cgi_to_mod_perl.html .

● **mod_perl_traps, common traps and solutions for mod_perl users**

at http://perl.apache.org/dist/mod_perl_traps.html .

● **mod_perl Quick Reference Card**

<http://www.refcards.com> (Apache and other refcards are available from this link)

● **mod_perl Resources Page**

http://www.perlreference.com/mod_perl/

● **mod_perl mailing list**

The Apache/Perl mailing list (modperl@apache.org) is available for mod_perl users and developers to share ideas, solve problems and discuss things related to mod_perl and the Apache::* modules. To subscribe to this list, send mail to modperl-subscribe@apache.org with empty

Subject and with Body:

```
subscribe modperl
```

A **searchable** mod_perl mailing list archive available at <http://forum.swarthmore.edu/epigone/modperl> . We owe it to Ken Williams.

More archives available:

- <http://www.geocrawler.com/lists/3/web/182/0/>
- <http://www.bitmechanic.com/mail-archives/modperl/>
- <http://www.mail-archive.com/modperl%40apache.org/>
- <http://www.davin.ottawa.on.ca/archive/modperl/>
- <http://www.progressive-comp.com/Lists/?l=apache-modperl&r=1&w=2#apache-modperl>
- <http://www.egroups.com/group/modperl/>

4.4 Get help with Perl

- **The Perl FAQ**

<http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html>

- **The Perl home**

<http://www.perl.com/>

- **The Perl Journal**

<http://www.tpj.com/>

- **Perl Module Mechanics**

http://world.std.com/~swmcd/steven/perl/module_mechanics.html - This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

4.5 Get help with Perl/CGI

- **Perl/CGI FAQ**

at <http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html>

- **Answers to some bothering Perl and Perl/CGI questions**

<http://stason.org/TULARC/webmaster/myfaq.html>

- **Idiot's Guide to CGI programming**

<http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html>

- **WWW Security FAQ**

<http://www.w3.org/Security/Faq/www-security-faq.html>

- **CGI/Perl Taint Mode FAQ**

<http://www.gunther.web66.com/FAQS/taintmode.html> (by Gunther Birznieks)

4.6 Get help with Apache

- **Apache Project's Home**

<http://www.apache.org>

- **Apache Quick Reference Card**

<http://www.refcards.com> (Apache and other refcards are available from this link)

- **The Apache FAQ**

<http://www.apache.org/docs/misc/FAQ.html>

- **Apache Server Documentation**

<http://www.apache.org/docs/>

- **Apache Handlers**

<http://www.apache.org/docs/handler.html>

- **mod_rewrite Guide**

<http://www.engelschall.com/pw/apache/rewriteguide/>

4.7 Get help with DBI

- **Perl DBI examples**

<http://www.saturn5.com/~jwb/dbi-examples.html> (by Jeffrey William Baker).

- **DBI Homepage**

<http://www.symbolstone.org/technology/perl/DBI/>

- **DBI mailing list information**

<http://www.fugue.com/dbi/>

- **DBI mailing list archives**

<http://outside.organic.com/mail-archives/dbi-users/> <http://www.xray.mpe.mpg.de/mailling-lists/dbi/>

- **Persistent connections with mod_perl**

http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS

4.8 Get help with Squid - Internet Object Cache

- Home page - <http://squid.nlanr.net/>
- FAQ - <http://squid.nlanr.net/Squid/FAQ/FAQ.html>
- Users Guide - <http://squid.nlanr.net/Squid/Users-Guide/>
- Mailing lists - <http://squid.nlanr.net/Squid/mailling-lists.html>

;o)

Table of Contents:

Tutorial: Improving Script Performance Under mod_perl	1
mod_perl tutorial: Getting Started Fast	3
1 Getting Started Fast	3
1.1 mod_perl in Four Slides	4
1.2 What is mod_perl?	4
1.3 Installation	5
1.4 Configuration	6
1.5 The "mod_perl rules" Apache::Registry Scripts	6
1.6 The "mod_perl rules" Apache Perl Module	7
1.7 Is That All I Need To Know About mod_perl?	7
mod_perl tutorial: Performance. Benchmarks.	9
2 Performance. Benchmarks.	9
2.1 What we will learn in this chapter	10
2.2 Performance: An Overall picture	11
2.3 Analysis of SW and HW Requirements	11
2.4 Sharing Memory	12
2.5 How Shared My Memory Is	12
2.6 Preload Perl modules at server startup	13
2.6.1 Preload Perl modules - Real Numbers	13
2.7 Preload Registry Scripts	17
2.8 Global vs Fully Qualified Variables	18
2.9 PerlSetupEnv Off	18
2.10 Adding a Proxy Server in http Accelerator Mode	19
2.11 KeepAlive	21
2.12 Upload/Download of Big Files	21
2.13 Forking or Executing subprocesses from mod_perl	22
2.14 Memory leakage	26
2.15 Checking script modification times	28
2.16 Cached stat() calls	28
2.17 Be carefull with symbolic links	29
2.18 Limiting the size of the processes	29
2.19 Limiting the resources used by httpd children	30
2.20 Limiting the request rate speed (robots blocking)	30
2.21 Benchmarks. Impressing your Boss and Colleagues.	31
2.21.1 Benchmarking scripts with execution times below 1 second :)	31
2.21.2 PerlHandler's Benchmarking	31
2.22 Tuning the Apache's configuration variables for the best performance	31
2.22.1 Tuning with ab - ApacheBench	32
2.22.2 Tuning with httpperf	37
2.22.3 Tuning with crashme script	38
2.22.4 Choosing MaxClients	41
2.22.5 Choosing MaxRequestsPerChild	43
2.22.6 Choosing MinSpareServers, MaxSpareServers and StartServers	43
2.22.7 Summary of Benchmarking to tune all 5 parameters	44

2.23	Persistent DB Connections	45
2.23.1	Preopening Connections at the Child Process' Fork Time	46
2.23.2	Caching prepare() statements	46
2.23.3	Handling Timeouts	47
2.24	Using \$ =1 under mod_perl and better print() techniques.	47
2.25	Avoid Importing Functions	49
2.26	Object Methods Calls Versus Function Calls	49
2.26.1	The Overhead with Light Subroutines	49
2.26.2	The Overhead with Heavy Subroutines	50
2.26.3	Are All Methods Slower than Functions?	51
2.26.4	Imported Symbols and Memory Usage	52
2.27	Sending plain HTML as a compressed output	53
	mod_perl tutorial: Choosing an Operating System and Hardware	55
3	Choosing an Operating System and Hardware	55
3.1	What we will learn in this chapter	56
3.2	Is it important?	56
3.3	Choosing an Operating System	56
3.3.1	Stability and Robustness	56
3.3.2	Memory Management	56
3.3.3	Memory Leakages	57
3.3.4	Sharing Memory	57
3.3.5	Cost and Support	57
3.3.6	Discontinued products	57
3.3.7	OS Releases	58
3.4	Choosing Hardware	58
3.4.1	Expected site traffic	59
3.4.2	Cash	59
3.4.3	Internet Connection	60
3.4.4	I/O performance	60
3.4.5	Memory	61
3.4.6	Bottlenecks	61
3.4.7	Conclusion	62
	mod_perl tutorial: Getting Help and Further Learning	63
4	Getting Help and Further Learning	63
4.1	What we will learn in this chapter	64
4.2	Getting help	64
4.3	Get help with mod_perl	64
4.4	Get help with Perl	66
4.5	Get help with Perl/CGI	66
4.6	Get help with Apache	67
4.7	Get help with DBI	67
4.8	Get help with Squid - Internet Object Cache	68