

1 Performance Tuning

1.1 Description

An exhaustive list of various techniques you might want to use to get the most performance possible out of your `mod_perl` server: configuration, coding, memory use and more.

1.2 The Big Picture

To make the user's Web browsing experience as painless as possible, every effort must be made to wring the last drop of performance from the server. There are many factors which affect Web site usability, but speed is one of the most important. This applies to any webserver, not just Apache, so it is very important that you understand it.

How do we measure the speed of a server? Since the user (and not the computer) is the one that interacts with the Web site, one good speed measurement is the time elapsed between the moment when she clicks on a link or presses a *Submit* button to the moment when the resulting page is fully rendered.

The requests and replies are broken into packets. A request may be made up of several packets, a reply may be many thousands. Each packet has to make its own way from one machine to another, perhaps passing through many interconnection nodes. We must measure the time starting from when the first packet of the request leaves our user's machine to when the last packet of the reply arrives back there.

A webserver is only one of the entities the packets see along their way. If we follow them from browser to server and back again, they may travel by different routes through many different entities. Before they are processed by your server the packets might have to go through proxy (accelerator) servers and if the request contains more than one packet, packets might arrive to the server by different routes with different arrival times, therefore it's possible that some packets that arrive earlier will have to wait for other packets before they could be reassembled into a chunk of the request message that will be then read by the server. Then the whole process is repeated in reverse.

You could work hard to fine tune your webserver's performance, but a slow Network Interface Card (NIC) or a slow network connection from your server might defeat it all. That's why it's important to think about the Big Picture and to be aware of possible bottlenecks between the server and the Web.

Of course there is little that you can do if the user has a slow connection. You might tune your scripts and webserver to process incoming requests ultra quickly, so you will need only a small number of working servers, but you might find that the server processes are all busy waiting for slow clients to accept their responses.

But there are techniques to cope with this. For example you can deliver the respond after it was compressed. If you are delivering a pure text respond--gzip compression will sometimes reduce the size of the respond by 10 times.

You should analyze all the involved components when you try to create the best service for your users, and not the web server or the code that the web server executes. A Web service is like a car, if one of the parts or mechanisms is broken the car may not go smoothly and it can even stop dead if pushed too far without first fixing it.

And let me stress it again--if you want to have a success in the web service business you should start worrying about the client's browsing experience and **not only** how good your code benchmarks are.

1.3 System Analysis

Before we try to solve a problem we need to identify it. In our case we want to get the best performance we can with as little monetary and time investment as possible.

1.3.1 *Software Requirements*

Covered in the section "Choosing an Operating System".

1.3.2 *Hardware Requirements*

(META: Only partial analysis. Please submit more points. Many points are scattered around the document and should be gathered here, to represent the whole picture. It also should be merged with the above item!)

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?
- At what rate are you expecting to receive requests?

The first one is probably the easiest to optimize. Following the performance optimization tips in this and other documents allows a perl (mod_perl) programmer to exercise their code and improve it.

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

Also ask yourself whether switching to another language will even help. In some applications, for example to link Oracle runtime libraries, a huge chunk of memory is needed so you would save nothing even if you switched from Perl to C.

The last two are important. You need a realistic estimate. 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 these numbers might change drastically when you apply code changes and your site becomes popular. Remember that when you get a very high hit rate, the resource requirements don't grow linearly but exponentially!

More coverage is provided in the section "Choosing Hardware".

1.4 Essential Tools

In order to improve performance we need measurement tools. The main tool categories are benchmarking and code profiling.

It's important to understand that in a major number of the benchmarking tests that we will execute we will not look at the absolute result numbers but the relation between the two and more result sets, since in most cases we would try to show which coding approach is preferable and the you shouldn't try to compare the absolute results collected while running the same benchmarks on your machine, since you won't have the exact hardware and software setup anyway. So this kind of comparison would be misleading. Compare the relative results from the tests running on your machine, don't compare your absolute results with those in this Guide.

1.4.1 *Benchmarking Applications*

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. Check out the `benchmark` directory of the `mod_perl` distribution for more examples.

If you are going to write your own benchmarking utility, use the `Benchmark` module for heavy scripts and the `Time::HiRes` module for very fast scripts (faster than 1 sec) where you will 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 the `LWP` package to emulate the browser. The `benchmark` directory of the `mod_perl` distribution includes such an example.

See also two tools for benchmarking: `ApacheBench` and `crashme` test

1.4.1.1 Benchmarking Perl Code

If you are going to write your own benchmarking utility, use the `Benchmark` module and the `Time::HiRes` module where you need better time precision (<10msec).

An example of the `Benchmark.pm` module usage:

```
benchmark.pl
-----
use Benchmark;

timethis (1_000,
  sub {
    my $x = 100;
    my $y = log ($x ** 100) for (0..10000);
  });
```

```
% perl benchmark.pl
timethis 1000: 25 wallclock secs (24.93 usr + 0.00 sys = 24.93 CPU)
```

If you want to get the benchmark results in micro-seconds you will have to use the `Time::HiRes` module, its usage is similar to `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 seconds."
```

See also the `crashme` test.

1.4.1.2 Benchmarking a Graphic Hits Counter with Persistent DB Connections

Here are the numbers from Michael Parker's `mod_perl` presentation at the Perl Conference (Aug, 98). (Sorry, there used to be links here to the source, but they went dead one day, so I removed them). The script is a standard hits counter, but it logs the counts into a mysql relational DataBase:

```
Benchmark: timing 100 iterations of cgi, perl... [rate 1:28]

cgi: 56 secs ( 0.33 usr 0.28 sys = 0.61 cpu)
perl: 2 secs ( 0.31 usr 0.27 sys = 0.58 cpu)

Benchmark: timing 1000 iterations of cgi,perl... [rate 1:21]

cgi: 567 secs ( 3.27 usr 2.83 sys = 6.10 cpu)
perl: 26 secs ( 3.11 usr 2.53 sys = 5.64 cpu)

Benchmark: timing 10000 iterations of cgi, perl [rate 1:21]

cgi: 6494 secs (34.87 usr 26.68 sys = 61.55 cpu)
perl: 299 secs (32.51 usr 23.98 sys = 56.49 cpu)
```

We don't know what server configurations were used for these tests, but I guess the numbers speak for themselves.

The source code of the script was available at <http://www.realtime.net/~parkerm/perl/conf98/sld006.htm>. It's now a dead link. If you know its new location, please let me know.

1.4.1.3 Benchmarking Response Times

In the next sections we will talk about tools that allow us to benchmark response times.

1.4.1.3.1 *ApacheBench*

`ApacheBench` (`ab`) is a tool for benchmarking your Apache HTTP server. It is designed to give you an idea of the performance that your current Apache installation can give. In particular, it shows you how many requests per second your Apache server is capable of serving. The `ab` tool comes bundled with the Apache source distribution.

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

```
% ./ab -n 100 -c 10 www.example.com/perl/test.pl
```

The results are:

```
Document Path:      /perl/test.pl
Document Length:    319 bytes

Concurrency Level:   10
Time taken for tests: 0.715 seconds
Complete requests:   100
Failed requests:     0
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
```

We can see that under load of ten concurrent users our server is capable of processing 140 requests per second. Of course this benchmark is correct only when the script under test is used. We can also learn about the average processing time, which in this case was 67 milli-seconds. Other numbers reported by `ab` may or may not be of interest to you.

For example if we believe that the script `perl/test.pl` is not efficient we will try to improve it and run the benchmark again, to see whether we have any improve in performance.

`HTTPD::Bench::ApacheBench`, available from CPAN, provides a Perl interface for `ab`.

1.4.1.3.2 *httperf*

`httperf` is a utility written by David Mosberger. Just like `ApacheBench`, it measures the performance of the webserver.

A sample command line is shown below:

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

This command causes `httperf` 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), independently of what load the server can actually sustain. 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

CPU time [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
```

httperf download

1.4.1.3.3 *http_load*

http_load is yet another utility that does webserver load testing. It can simulate 33.6kbps modem connection (*-throttle*) and allows you to provide a file with a list of URLs, which we be fetched randomly. You can specify how many parallel connections to run using the *-parallel N* option, or you can specify the number of requests to generate per second with *-rate N* option. Finally you can tell the utility when to stop by specifying either the test time length (*-seconds N*) or the total number of fetches (*-fetches N*).

A sample run with the file *urls* including:

```
http://www.example.com/foo/
http://www.example.com/bar/
```

We ask to generate three requests per second and run for only two seconds. Here is the generated output:

```
% ./http_load -rate 3 -seconds 2 urls
http://www.example.com/foo/: check-connect SUCCEEDED, ignoring
http://www.example.com/bar/: check-connect SUCCEEDED, ignoring
http://www.example.com/bar/: check-connect SUCCEEDED, ignoring
http://www.example.com/bar/: check-connect SUCCEEDED, ignoring
http://www.example.com/foo/: check-connect SUCCEEDED, ignoring
5 fetches, 3 max parallel, 96870 bytes, in 2.00258 seconds
19374 mean bytes/connection
2.49678 fetches/sec, 48372.7 bytes/sec
msecs/connect: 1.805 mean, 5.24 max, 0.79 min
msecs/first-response: 291.289 mean, 560.338 max, 34.349 min
```

So you can see that it has reported 2.5 requests per second. Of course for the real test you will want to load the server heavily and run the test for a longer time to get more reliable results.

Note that when you provide a file with a list of URLs make sure that you don't have empty lines in it. If you do -- the utility won't work complaining:

```
./http_load: unknown protocol -
```

http_load download

1.4.1.3.4 the crashme Script

This is another crashme suite originally written by Michael Schilli (and was located at <http://www.linux-magazin.de> site, but now the link has gone). I made a few modifications, mostly adding my() operators. I also allowed it to accept more than one url to test, since sometimes you want to test more than 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 values for **Latency** (seconds per request) and **Throughput** (requests per second). It can do a complete simulation of your favorite Netscape browser :) and give you a better picture.

I have noticed while running these two benchmarking suites, that **ab** gave me results from two and a half to three times better. Both suites were run on the same machine, with the same load and the same parameters, but the implementations were different.

Sample output:

```
URL(s):      http://www.example.com/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:

The LWP::Parallel::UserAgent benchmark: *code/lwp-bench.pl*:

```
#!/usr/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.example.com:81/perl/faq_manager/faq_manager.pl',
    'http://www.example.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
###

```

1.4.1 Benchmarking Applications

```
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;
my @P = (
  "URL(s)"          => join("\n\t\t", @urls),
  "Total Requests"  => $nof_requests_total * @urls,
  "Parallel Agents" => $nof_parallel_connections,
  "Succeeded"       => sprintf("$succeeded (%.2f%%)\n",
                              $succeeded * 100 / ( $nof_requests_total * @urls ) ),
  "Errors"          => $errors,
  "Total Time"      => sprintf("%.2f secs\n", $total_time),
  "Throughput"      => sprintf("%.2f Requests/sec\n",
                              ( $nof_requests_total * @urls ) / $total_time),
  "Latency"         => sprintf("%.2f secs/Request",
                              ( $ua->get_latency_total() || 0 ) /
                              ( $nof_requests_total * @urls ) ),
);
```

```
my ($left, $right);  
###  
# Print out statistics  
###  
format STDOUT =  
@<<<<<<<<<<<<<< @*  
"$left:",           $right  
.  
  
while(($left, $right) = splice(@P, 0, 2)) {  
    write;  
}
```

1.4.1.4 Benchmarking PerlHandlers

The `Apache::Timeit` module does `PerlHandler` Benchmarking. With the help of this module you can log the time taken to process the request, just like you'd use the `Benchmark` module to benchmark a regular Perl script. Of course you can extend this module to perform more advanced processing like putting the results into a database for a later processing. But all it takes is adding this configuration directive inside *httpd.conf*:

```
PerlFixupHandler Apache::Timeit
```

Since scripts running under `Apache::Registry` are running inside the `PerlHandler` these are benchmarked as well.

An example of the lines which show up in the *error_log* file:

```

timing request for /perl/setupenvoff.pl:
  0 wallclock secs ( 0.04 usr +  0.01 sys =  0.05 CPU)
timing request for /perl/setupenvoff.pl:
  0 wallclock secs ( 0.03 usr +  0.00 sys =  0.03 CPU)

```

The `Apache::Timeit` package is a part of the *Apache-Perl-contrib* files collection available from CPAN.

1.4.1.5 Other Benchmarking Tools

Other tools you may want to take a look at:

- HTTP::WebTest

HTTP::WebTest module runs tests on remote URLs or local web files containing Perl/JSP/HTML/JavaScript/etc. and generates a detailed test report.

It's available from CPAN.

- **HTTP::Monkeywrench**

HTTP: :Monkeywrench is a test-harness application to test the integrity of a user's path through a web site.

It's available from CPAN.

- **Apache::Recorder and HTTP::RecordedSession**

Apache::Recorder is a mod_perl handler that records an HTTP session and stores it on the web server's file system. HTTP::RecordedSession reads the recorded session from the file system, and formats it for playback using HTTP::WebTest or HTTP::Monkeywrench. This is useful when writing acceptance and regression tests.

It's available from CPAN.

1.4.2 Code Profiling Techniques

The profiling process helps you to determine which subroutines or just snippets of code take the longest time to execute and which subroutines are called most often. Probably you will want to optimize those.

When do you need to profile your code? You do that when you suspect that some part of your code is called very often and may be there is a need to optimize it to significantly improve the overall performance.

For example if you have ever used the `diagnostics` pragma, which extends the terse diagnostics normally emitted by both the Perl compiler and the Perl interpreter, augmenting them with the more verbose and endearing descriptions found in the `perldiag` manpage. You know that it might tremendously slow you code down, so let's first prove that it is correct.

We will run a benchmark, once with diagnostics enabled and once disabled, on a subroutine called *test_code*.

The code inside the subroutine does an arithmetic and a numeric comparison of two strings. It assigns one string to another if the condition tests true but the condition always tests false. To demonstrate the `diagnostics` overhead the comparison operator is intentionally *wrong*. It should be a string comparison, not a numeric one.

```
use Benchmark;
use diagnostics;
use strict;

my $count = 50000;

disable diagnostics;
my $t1 = timeit($count,\&test_code);

enable diagnostics;
my $t2 = timeit($count,\&test_code);

print "Off: ",timestr($t1),"\n";
print "On : ",timestr($t2),"\n";

sub test_code{
    my ($a,$b) = qw(foo bar);
    my $c;
```

```

    if ($a == $b) {
        $c = $a;
    }
}

```

For only a few lines of code we get:

```

Off:  1 wallclock secs ( 0.81 usr +  0.00 sys =  0.81 CPU)
On  : 13 wallclock secs (12.54 usr +  0.01 sys = 12.55 CPU)

```

With `diagnostics` enabled, the subroutine `test_code()` is 16 times slower, than with `diagnostics` disabled!

Now let's fix the comparison the way it should be, by replacing `==` with `eq`, so we get:

```

my ($a,$b) = qw(foo bar);
my $c;
if ($a eq $b) {
    $c = $a;
}

```

and run the same benchmark again:

```

Off:  1 wallclock secs ( 0.57 usr +  0.00 sys =  0.57 CPU)
On  :  1 wallclock secs ( 0.56 usr +  0.00 sys =  0.56 CPU)

```

Now there is no overhead at all. The `diagnostics` pragma slows things down only when warnings are generated.

After we have verified that using the `diagnostics` pragma might adds a big overhead to execution runtime, let's use the code profiling to understand why this happens. We are going to use `Devel::DProf` to profile the code. Let's use this code:

```

diagnostics.pl
-----
use diagnostics;
print "Content-type:text/html\n\n";
test_code();
sub test_code{
    my ($a,$b) = qw(foo bar);
    my $c;
    if ($a == $b) {
        $c = $a;
    }
}

```

Run it with the profiler enabled, and then create the profiling stastics with the help of `dprofpp`:

```

% perl -d:DProf diagnostics.pl
% dprofpp

Total Elapsed Time = 0.342236 Seconds
User+System Time = 0.335420 Seconds
Exclusive Times

```

1.4.2 Code Profiling Techniques

%Time	ExclSec	CumulS	#Calls	sec/call	Csec/c	Name
92.1	0.309	0.358	1	0.3089	0.3578	main::BEGIN
14.9	0.050	0.039	3161	0.0000	0.0000	diagnostics::unescape
2.98	0.010	0.010	2	0.0050	0.0050	diagnostics::BEGIN
0.00	0.000	-0.000	2	0.0000	-	Exporter::import
0.00	0.000	-0.000	2	0.0000	-	Exporter::export
0.00	0.000	-0.000	1	0.0000	-	Config::BEGIN
0.00	0.000	-0.000	1	0.0000	-	Config::TIEHASH
0.00	0.000	-0.000	2	0.0000	-	Config::FETCH
0.00	0.000	-0.000	1	0.0000	-	diagnostics::import
0.00	0.000	-0.000	1	0.0000	-	main::test_code
0.00	0.000	-0.000	2	0.0000	-	diagnostics::warn_trap
0.00	0.000	-0.000	2	0.0000	-	diagnostics::splainthis
0.00	0.000	-0.000	2	0.0000	-	diagnostics::transmo
0.00	0.000	-0.000	2	0.0000	-	diagnostics::shorten
0.00	0.000	-0.000	2	0.0000	-	diagnostics::autodescribe

It's not easy to see what is responsible for this enormous overhead, even if `main::BEGIN` seems to be running most of the time. To get the full picture we must see the OPs tree, which shows us who calls whom, so we run:

```
% dprofpp -T
```

and the output is:

```
main::BEGIN
  diagnostics::BEGIN
    Exporter::import
    Exporter::export
  diagnostics::BEGIN
    Config::BEGIN
    Config::TIEHASH
    Exporter::import
    Exporter::export
  Config::FETCH
  Config::FETCH
  diagnostics::unescape
  .....
  3159 times [diagnostics::unescape] snipped
  .....
  diagnostics::unescape
  diagnostics::import
diagnostics::warn_trap
  diagnostics::splainthis
    diagnostics::transmo
    diagnostics::shorten
    diagnostics::autodescribe
main::test_code
  diagnostics::warn_trap
    diagnostics::splainthis
      diagnostics::transmo
      diagnostics::shorten
      diagnostics::autodescribe
  diagnostics::warn_trap
```

```

diagnostics::splainthis
diagnostics::transmo
diagnostics::shorten
diagnostics::autodescribe

```

So we see that two executions of `diagnostics::BEGIN` and 3161 of `diagnostics::unescape` are responsible for most of the running overhead.

If we comment out the `diagnostics` module, we get:

```

Total Elapsed Time = 0.079974 Seconds
User+System Time = 0.059974 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
0.00 0.000 -0.000 1 0.0000 - main::test_code

```

It is possible to profile code running under `mod_perl` with the `Devel::DProf` module, available on CPAN. However, you must have apache version 1.3b3 or higher and the `PerlChildExitHandler` enabled during the `httpd` build process. When the server is started, `Devel::DProf` installs an `END` block to write the *tmon.out* file. This block will be called at server shutdown. Here is how to start and stop a server with the profiler enabled:

```

% setenv PERL5OPT -d:DProf
% httpd -X -d 'pwd' &
... make some requests to the server here ...
% kill 'cat logs/httpd.pid'
% unsetenv PERL5OPT
% dprofpp

```

The `Devel::DProf` package is a Perl code profiler. It will collect information on the execution time of a Perl script and of the subs in that script (remember that `print()` and `map()` are just like any other subroutines you write, but they come bundled with Perl!)

Another approach is to use `Apache::DProf`, which hooks `Devel::DProf` into `mod_perl`. The `Apache::DProf` module will run a `Devel::DProf` profiler inside each child server and write the *tmon.out* file in the directory `$ServerRoot/logs/dprof/$$` when the child is shutdown (where `$$` is the number of the child process). All it takes is to add to *httpd.conf*:

```
PerlModule Apache::DProf
```

Remember that any `PerlHandler` that was pulled in before `Apache::DProf` in the *httpd.conf* or *startup.pl*, will not have its code debugging information inserted. To run `dprofpp`, `chdir` to `$ServerRoot/logs/dprof/$$` and run:

```
% dprofpp
```

(Lookup the `ServerRoot` directive's value in *httpd.conf* to figure out what's your `$ServerRoot`.)

1.4.3 Measuring the Memory of the Process

Very important aspect of performance tuning is to make sure that your applications don't use much memory, since if they do you cannot run many servers and therefore in most cases under a heavy load the overall performance degrades.

In addition the code may not be clean and leak memory, which is even worse, since if the same process serves many requests and after each request more memory is used, after awhile all RAM will be used and machine will start swapping (use the swap partition) which is a very undesirable event, since it may lead to a machine crash.

The simplest way to figure out how big the processes are and see whether they grow is to watch the output of `top(1)` or `ps(1)` utilities.

For example the output of `top(1)`:

```
8:51am up 66 days, 1:44, 1 user, load average: 1.09, 2.27, 2.61
95 processes: 92 sleeping, 3 running, 0 zombie, 0 stopped
CPU states: 54.0% user, 9.4% system, 1.7% nice, 34.7% idle
Mem: 387664K av, 309692K used, 77972K free, 111092K shrd, 70944K buff
Swap: 128484K av, 11176K used, 117308K free, 170824K cached
```

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	LIB	%CPU	%MEM	TIME	COMMAND
29225	nobody	0	0	9760	9760	7132	S	0	12.5	2.5	0:00	httpd_perl
29220	nobody	0	0	9540	9540	7136	S	0	9.0	2.4	0:00	httpd_perl
29215	nobody	1	0	9672	9672	6884	S	0	4.6	2.4	0:01	httpd_perl
29255	root	7	0	1036	1036	824	R	0	3.2	0.2	0:01	top
376	squid	0	0	15920	14M	556	S	0	1.1	3.8	209:12	squid
29227	mysql	5	5	1892	1892	956	S N	0	1.1	0.4	0:00	mysqld
29223	mysql	5	5	1892	1892	956	S N	0	0.9	0.4	0:00	mysqld
29234	mysql	5	5	1892	1892	956	S N	0	0.9	0.4	0:00	mysqld

Which starts with overall information of the system and then displays the most active processes at the given moment. So for example if we look at the `httpd_perl` processes we can see the size of the resident (RSS) and shared (SHARE) memory segments. This sample was taken on the production server running linux.

But of course we want to see all the `apache/mod_perl` processes, and that's where `ps(1)` comes to help. The options of this utility vary from one Unix flavor to another, and some flavors provide their own tools. Let's check the information about `mod_perl` processes:

```
% ps -o pid,user,rss,vsize,%cpu,%mem,ucomm -C httpd_perl
  PID USER      RSS   VSZ %CPU %MEM COMMAND
 29213 root        8584 10264 0.0  2.2 httpd_perl
 29215 nobody     9740 11316 1.0  2.5 httpd_perl
 29216 nobody     9668 11252 0.7  2.4 httpd_perl
 29217 nobody     9824 11408 0.6  2.5 httpd_perl
 29218 nobody     9712 11292 0.6  2.5 httpd_perl
 29219 nobody     8860 10528 0.0  2.2 httpd_perl
 29220 nobody     9616 11200 0.5  2.4 httpd_perl
 29221 nobody     8860 10528 0.0  2.2 httpd_perl
```



```

29222 nobody    8860 10528  0.0  2.2 httpd_perl
29224 nobody    8860 10528  0.0  2.2 httpd_perl
29225 nobody    9760 11340  0.7  2.5 httpd_perl
29235 nobody    9524 11104  0.4  2.4 httpd_perl

```

Now you can see the resident (RSS) and virtual (VSZ) memory segments (and shared memory segment if you ask for it) of all mod_perl processes. Please refer to the top(1) and ps(1) man pages for more information.

You probably agree that using top(1) and ps(1) is cumbersome if we want to use memory size sampling during the benchmark test. We want to have a way to print memory sizes during the program execution at desired places. If you have GTop modules installed, which is a perl glue to the libgtop library, it's exactly what we need.

Note: GTop requires the libgtop library but is not available for all platforms. See the docs in the source at <ftp://ftp.gnome.org/pub/GNOME/stable/sources/gtop/> to check whether your platform/flavor is supported.

GTop provides an API for retrieval of information about processes and the whole system. We are interested only in memory sampling API methods. To print all the process related memory information we can execute the following code:

```

use GTop;
my $gtop = GTop->new;
my $proc_mem = $gtop->proc_mem($$);
for (qw(size vsize share rss)) {
    printf "    %s => %d\n", $_, $proc_mem->$_();
}

```

When executed we see the following output (in bytes):

```

size => 1900544
vsize => 3108864
share => 1392640
rss => 1900544

```

So if we are interested in to print the process resident memory segment before and after some event we just do it: For example if we want to see how much extra memory was allocated after a variable creation we can write the following code:

```

use GTop;
my $gtop = GTop->new;
my $before = $gtop->proc_mem($$)->rss;
my $x = 'a' x 10000;
my $after = $gtop->proc_mem($$)->rss;
print "diff: ", $after-$before, " bytes\n";

```

and the output

```
diff: 20480 bytes
```

So we can see that Perl has allocated extra 20480 bytes to create `$x` (of course the creation of `after` needed a few bytes as well, but it's insignificant compared to a size of `$x`)

The `Apache::VMonitor` module with help of the `GTop` module allows you to watch all your system information using your favorite browser from anywhere in the world without a need to telnet to your machine. If you are looking at what information you can retrieve with `GTop`, you should look at `Apache::VMonitor` as it deploys a big part of the API `GTop` provides.

If you are running a true BSD system, you may use `BSD::Resource::getrusage` instead of `GTop`. For example:

```
print "used memory = " . (BSD::Resource::getrusage)[2] . "\n"
```

For more information refer to the `BSD::Resource` manpage.

1.4.4 Measuring the Memory Usage of Subroutines

With help of `Apache::Status` you can find out the size of each and every subroutine.

1. Build and install `mod_perl` as you always do, make sure it's version 1.22 or higher.
2. Configure `/perl-status` if you haven't already:

```
<Location /perl-status>
    SetHandler perl-script
    PerlHandler Apache::Status
    order deny,allow
    #deny from all
    #allow from ...
</Location>
```

3. Add to `httpd.conf`

```
PerlSetVar StatusOptionsAll On
PerlSetVar StatusTerse On
PerlSetVar StatusTerseSize On
PerlSetVar StatusTerseSizeMainSummary On

PerlModule B::TerseSize
```

4. Start the server (best in `httpd -X` mode)
5. From your favorite browser fetch `http://localhost/perl-status`
6. Click on 'Loaded Modules' or 'Compiled Registry Scripts'
7. Click on the module or script of your choice (you might need to run some script/handler before you will see it here unless it was preloaded)

8. Click on 'Memory Usage' at the bottom
9. You should see all the subroutines and their respective sizes.

Now you can start to optimize your code. Or test which of the several implementations is of the least size.

For example let's compare CGI.pm's OO vs. procedural interfaces:

As you will see below the first OO script uses about 2k bytes while the second script (procedural interface) uses about 5k.

Here are the code examples and the numbers:

1.

```
cgi_oo.pl
-----
use CGI ();
my $q = CGI->new;
print $q->header;
print $q->b("Hello");
```

2.

```
cgi_mtd.pl
-----
use CGI qw(header b);
print header();
print b("Hello");
```

After executing each script in single server mode (-X) the results are:

1.

```
Totals: 1966 bytes | 27 OPs

handler 1514 bytes | 27 OPs
exit     116 bytes |  0 OPs
```

2.

```
Totals: 4710 bytes | 19 OPs

handler 1117 bytes | 19 OPs
basefont 120 bytes |  0 OPs
frameset 120 bytes |  0 OPs
caption  119 bytes |  0 OPs
applet   118 bytes |  0 OPs
script   118 bytes |  0 OPs
ilayer   118 bytes |  0 OPs
header   118 bytes |  0 OPs
strike   118 bytes |  0 OPs
layer    117 bytes |  0 OPs
table    117 bytes |  0 OPs
frame    117 bytes |  0 OPs
style    117 bytes |  0 OPs
```

1.4.4 Measuring the Memory Usage of Subroutines

Param	117 bytes		0 OPs
small	117 bytes		0 OPs
embed	117 bytes		0 OPs
font	116 bytes		0 OPs
span	116 bytes		0 OPs
exit	116 bytes		0 OPs
big	115 bytes		0 OPs
div	115 bytes		0 OPs
sup	115 bytes		0 OPs
Sub	115 bytes		0 OPs
TR	114 bytes		0 OPs
td	114 bytes		0 OPs
Tr	114 bytes		0 OPs
th	114 bytes		0 OPs
b	113 bytes		0 OPs

Note, that the above is correct if you didn't precompile all CGI.pm's methods at server startup. Since if you did, the procedural interface in the second test will take up to 18k and not 5k as we saw. That's because the whole of CGI.pm's namespace is inherited and it already has all its methods compiled, so it doesn't really matter whether you attempt to import only the symbols that you need. So if you have:

```
use CGI qw(-compile :all);
```

in the server startup script. Having:

```
use CGI qw(header);
```

or

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

is essentially the same. You will have all the symbols precompiled at startup imported even if you ask for only one symbol. It seems to me like a bug, but probably that's how CGI.pm works.

BTW, you can check the number of opcodes in the code by a simple command line run. For example comparing 'my %hash' vs. 'my %hash = ()'.

```
% perl -MO=Terse -e 'my %hash' | wc -l
-e syntax OK
4

% perl -MO=Terse -e 'my %hash = ()' | wc -l
-e syntax OK
10
```

The first one has less opcodes.

Note that you shouldn't use Apache::Status module on production server as it adds quite a bit of overhead for each request.

1.5 Know Your Operating System

In order to get the best performance it helps to get intimately familiar with the Operating System (OS) the web server is running on. There are many OS specific things that you may be able to optimize which will improve your web server's speed, reliability and security.

The following sections will reveal some of the most important details you should know about your OS.

1.5.1 *Sharing Memory*

The sharing of memory is one very important factor. If your OS supports it (and most sane systems do), you might save 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 tend to become unshared.

There is no way we can make Perl allocate memory so that (dynamic) variables land on different memory pages from constants, so the **copy-on-write** effect (we will explain this in a moment) will hit you almost at random.

If you are pre-loading many modules you might be able to trade off the memory that stays shared against the time for an occasional fork by tuning `MaxRequestsPerChild`. Each time a child reaches this upper limit and dies it should release its unshared pages. The new child which replaces it will share its fresh pages until it scribbles on them.

The ideal is a point where your processes usually restart before too much memory becomes unshared. You should take some measurements to see if it makes a real difference, and to find the range of reasonable values. If you have success with this tuning the value of `MaxRequestsPerChild` will probably be peculiar to your situation and may change with changing circumstances.

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 overall speedup, so if it is 100 or 10000 it probably does not really matter if you can save RAM by using a lower value.

Do not forget that if you preload most of your code at server startup, the newly forked child gets ready very fast, because it inherits most of the preloaded code and the perl interpreter from the parent process.

During the life of the child its memory pages (which aren't really its own to start with, it uses the parent's pages) gradually get 'dirty' - variables which were originally inherited and shared are updated or modified -- and the *copy-on-write* happens. This reduces the number of shared memory pages, thus increasing the memory requirement. Killing the child and spawning a new one allows the new child to get back to the pristine shared memory of the parent process.

The recommendation is that `MaxRequestsPerChild` should not be too large, otherwise you lose some of the benefit of sharing memory.

See [Choosing MaxRequestsPerChild](#) for more about tuning the `MaxRequestsPerChild` parameter.

1.5.1.1 How Shared Is My Memory?

You've probably noticed that the word `shared` is repeated many times in relation 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. See the Formula and the numbers.

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

```
use GTop ();
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 the `RES` (or `RSS`) columns with the `SHARE` column. `RES` is `RESident` memory, which is the size of pages currently swapped in.

1.5.1.2 Calculating Real Memory Usage

I have shown how to measure the size of the process' shared memory, but we still want to know what the real memory usage is. Obviously this cannot be calculated simply by adding up the memory size of each process because that wouldn't account for the shared memory.

On the other hand we cannot just subtract the shared memory size from the total size to get the real memory usage numbers, because in reality each process has a different history of processed requests, therefore the shared memory is not the same for all processes.

So how do we measure the real memory size used by the server we run? It's probably too difficult to give the exact number, but I've found a way to get a fair approximation which was verified in the following way. I have calculated the real memory used, by the technique you will see in the moment, and then have stopped the Apache server and saw that the memory usage report indicated that the total used memory went down by almost the same number I've calculated. Note that some OSs do smart memory pages caching so you may not see the memory usage decrease as soon as it actually happens when you quit the application.

This is a technique I've used:

1. For each process sum up the difference between shared and system memory. To calculate a difference for a single process use:

```
use GTop;
my $proc_mem = GTop->new->proc_mem($$);
my $diff     = $proc_mem->size - $proc_mem->share;
print "Difference is $diff bytes\n";
```

2. Now if we add the shared memory size of the process with maximum shared memory, we will get all the memory that actually is being used by all `httpd` processes, except for the parent process.

3. Finally, add the size of the parent process.

Please note that this might be incorrect for your system, so you use this number on your own risk.

I've used this technique to display real memory usage in the module `Apache::VMonitor`, so instead of trying to manually calculate this number you can use this module to do it automatically. In fact in the calculations used in this module there is no separation between the parent and child processes, they are all counted indifferently using the following code:

```
use GTop ();
my $gtop = GTop->new;
my $total_real = 0;
my $max_shared = 0;
# @mod_perl_pids is initialized by Apache::Scoreboard, irrelevant here
my @mod_perl_pids = some_code();
for my $pid (@mod_perl_pids)
{
    my $proc_mem = $gtop->proc_mem($pid);
    my $size      = $proc_mem->size($pid);
    my $share     = $proc_mem->share($pid);
    $total_real += $size - $share;
    $max_shared = $share if $max_shared < $share;
}
my $total_real += $max_shared;
```

So as you see we that we accumulate the difference between the shared and reported memory:

```
$total_real += $size-$share;
```

and at the end add the biggest shared process size:

```
my $total_real += $max_shared;
```

So now `$total_real` contains approximately the really used memory.

1.5.1.3 Are My Variables Shared?

How do you find out if the code you write is shared between the processes or not? The code should be shared, except where it is on a memory page with variables that change. Some variables are read-only in usage and never change. For example, if you have some variables that use a lot of memory and you want them to be read-only. As you know the variable becomes unshared when the process modifies its value.

So imagine that you have this 10Mb in-memory database that resides in a single variable, you perform various operations on it and want to make sure that the variable is still shared. For example if you do some matching regular expression (regex) processing on this variable and want to use the `pos()` function, will it make the variable unshared or not?

The `Apache::Peek` module comes to rescue. Let's write a module called *MyShared.pm* which we preload at server startup, so all the variables of this module are initially shared by all children.

1.5.1 Sharing Memory

```
MyShared.pm
-----
package MyShared;
use Apache::Peek;

my $readonly = "Chris";

sub match    { $readonly =~ /\w/g; }
sub print_pos { print "pos: ", pos($readonly), "\n"; }
sub dump     { Dump($readonly); }
1;
```

This module declares the package `MyShared`, loads the `Apache::Peek` module and defines the lexically scoped `$readonly` variable which is supposed to be a variable of large size (think about a huge hash data structure), but we will use a small one to simplify this example.

The module also defines three subroutines: `match()` that does a simple character matching, `print_pos()` that prints the current position of the matching engine inside the string that was last matched and finally the `dump()` subroutine that calls the `Apache::Peek` module's `Dump()` function to dump a raw Perl data-type of the `$readonly` variable.

Now we write the script that prints the process ID (PID) and calls all three functions. The goal is to check whether `pos()` makes the variable *dirty* and therefore unshared.

```
share_test.pl
-----
use MyShared;
print "Content-type: text/plain\r\n\r\n";
print "PID: $$\n";
MyShared::match();
MyShared::print_pos();
MyShared::dump();
```

Before you restart the server, in *httpd.conf* set:

```
MaxClients 2
```

for easier tracking. You need at least two servers to compare the print outs of the test program. Having more than two can make the comparison process harder.

Now open two browser windows and issue the request for this script several times in both windows, so you get different processes PIDs reported in the two windows and each process has processed a different number of requests to the *share_test.pl* script.

In the first window you will see something like that:

```
PID: 27040
pos: 1
SV = PVMG(0x853db20) at 0x8250e8c
  REFCNT = 3
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
```



```

PV = 0x8271af0 "Chris"\0
CUR = 5
LEN = 6
MAGIC = 0x853dd80
  MG_VIRTUAL = &vtbl_mglob
  MG_TYPE = 'g'
  MG_LEN = 1

```

And in the second window:

```

PID: 27041
pos: 2
SV = PVMG(0x853db20) at 0x8250e8c
  REFCNT = 3
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
  PV = 0x8271af0 "Chris"\0
  CUR = 5
  LEN = 6
  MAGIC = 0x853dd80
    MG_VIRTUAL = &vtbl_mglob
    MG_TYPE = 'g'
    MG_LEN = 2

```

We see that all the addresses of the supposedly big structure are the same (0x8250e8c and 0x8271af0), therefore the variable data structure is almost completely shared. The only difference is in SV.MAGIC.MG_LEN record, which is not shared.

So given that the \$readonly variable is a big one, its value is still shared between the processes, while part of the variable data structure is non-shared. But it's almost insignificant because it takes a very little memory space.

Now if you need to compare more than variable, doing it by hand can be quite time consuming and error prone. Therefore it's better to correct the testing script to dump the Perl data-types into files (e.g /tmp/dump.\$\$, where \$\$ is the PID of the process) and then using diff(1) utility to see whether there is some difference.

So correcting the dump() function to write the info to the file will do the job. Notice that we use Devel::Peek and not Apache::Peek. The both are almost the same, but Apache::Peek prints its output directly to the opened socket so we cannot intercept and redirect the result to the file. Since Devel::Peek dumps results to the STDERR stream we can use the old trick of saving away the default STDERR handler, and open a new filehandler using the STDERR. In our example when Devel::Peek now prints to STDERR it actually prints to our file. When we are done, we make sure to restore the original STDERR filehandler.

So this is the resulting code:

```

MyShared2.pm
-----
package MyShared2;
use Devel::Peek;

```

```

my $readonly = "Chris";

sub match    { $readonly =~ /\w/g; }
sub print_pos{ print "pos: ",pos($readonly),"\n";}
sub dump{
    my $dump_file = "/tmp/dump.$$";
    print "Dumping the data into $dump_file\n";
    open OLDERR, ">&STDERR";
    open STDERR, ">".$dump_file or die "Can't open $dump_file: $!";
    Dump($readonly);
    close STDERR ;
    open STDERR, ">&OLDERR";
}
1;

```

When if we modify the code to use the modified module:

```

share_test2.pl
-----
use MyShared2;
print "Content-type: text/plain\r\n\r\n";
print "PID: $$\n";
MyShared2::match();
MyShared2::print_pos();
MyShared2::dump();

```

And run it as before (with MaxClients 2), two dump files will be created in the directory */tmp*. In our test these were created as */tmp/dump.1224* and */tmp/dump.1225*. When we run `diff(1)`:

```

% diff /tmp/dump.1224 /tmp/dump.1225
12c12
<         MG_LEN = 1
---
>         MG_LEN = 2

```

We see that the two padlists (of the variable `readonly`) are different, as we have observed before when we did a manual comparison.

In fact we if we think about these results again, we get to a conclusion that there is no need for two processes to find out whether the variable gets modified (and therefore unshared). It's enough to check the datastructure before the script was executed and after that. You can modify the `MyShared2` module to dump the padlists into a different file after each invocation and than to run the `diff(1)` on the two files.

If you want to watch whether some lexically scoped (with `my()`) variables in your `Apache::Registry` script inside the same process get changed between invocations you can use the `Apache::RegistryLexInfo` module instead. Since it does exactly this: it makes a snapshot of the padlist before and after the code execution and shows the difference between the two. This specific module was written to work with `Apache::Registry` scripts so it won't work for loaded modules. Use the technique we have described above for any type of variables in modules and scripts.

Surely another way of ensuring that a scalar is readonly and therefore sharable is to either use the `constant` pragma or `readonly` pragma. But then you won't be able to make calls that alter the variable even a little, like in the example that we just showed, because it will be a true constant variable and you will get compile time error if you try this:

```
MyConstant.pm
-----
package MyConstant;
use constant readonly => "Chris";

sub match    { readonly =~ /\w/g; }
sub print_pos{ print "pos: ",pos(readonly),"\n"; }
1;

% perl -c MyConstant.pm

Can't modify constant item in match position at MyConstant.pm line
5, near "readonly)"
MyConstant.pm had compilation errors.
```

However this code is just right:

```
MyConstant1.pm
-----
package MyConstant1;
use constant readonly => "Chris";

sub match { readonly =~ /\w/g; }
1;
```

1.5.1.4 Preloading Perl Modules at Server Startup

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

```
use DBI ();
use Carp ();
```

Don't forget to prevent importing of the symbols exported by default by the module you are going to preload, by placing empty parentheses `()` after a module's name. Unless you need some of these in the startup file, which is unlikely. This will save you a few more memory bits.

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

1.5.1 Sharing Memory

```
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. When you use `mod_perl`, `FastCGI` or another system that uses a persistent Perl interpreter, you will want to precompile the functions 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 the tag names that you actually use in your code, since generally you only use a subset of them.

Let's conduct a memory usage test to prove that preloading, reduces memory requirements.

In order to have an easy measurement we will use only one child process, therefore we will use this setting:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We are going to use the `Apache::Registry` script *memuse.pl* which consists of two parts: the first one preloads a bunch of modules (that most of them aren't going to be used), the second part reports the memory size and the shared memory size used by the single child process that we start. and of course it prints the difference between the two sizes.

```
memuse.pl
-----
use strict;
use CGI ();
use DB_File ();
use LWP::UserAgent ();
use Storable ();
use DBI ();
use GTop ();

my $r = shift;
$r->send_http_header('text/plain');
my $proc_mem = GTop->new->proc_mem($$);
my $size = $proc_mem->size;
my $share = $proc_mem->share;
my $diff = $size - $share;
printf "%10s %10s %10s\n", qw(Size Shared Difference);
printf "%10d %10d %10d (bytes)\n", $size, $share, $diff;
```

First we restart the server and execute this CGI script when none of the above modules preloaded. Here is the result:

Size	Shared	Diff
4706304	2134016	2572288 (bytes)

Now we take all the modules:

```
use strict;
use CGI ();
use DB_File ();
use LWP::UserAgent ();
use Storable ();
use DBI ();
use GTop ();
```

and copy them into the startup script, so they will get preloaded. The script remains unchanged. We restart the server and execute it again. We get the following.

Size	Shared	Diff
4710400	3997696	712704 (bytes)

Let's put the two results into one table:

Preloading	Size	Shared	Diff
Yes	4710400	3997696	712704 (bytes)
No	4706304	2134016	2572288 (bytes)

Difference	4096	1863680	-1859584

You can clearly see that when the modules weren't preloaded the shared memory pages size, were about 1864Kb smaller relative to the case where the modules were preloaded.

Assuming that you have had 256M dedicated to the web server, if you didn't preload the modules, you could have:

$$268435456 = X * 2572288 + 2134016$$

$$X = (268435456 - 2134016) / 2572288 = 103$$

103 servers.

Now let's calculate the same thing with modules preloaded:

$$268435456 = X * 712704 + 3997696$$

$$X = (268435456 - 3997696) / 712704 = 371$$

You can have almost 4 times more servers!!!

Remember that we have mentioned before that memory pages gets dirty and the size of the shared memory gets smaller with time? So we have presented the ideal case where the shared memory stays intact. Therefore the real numbers will be a little bit different, but not far from the numbers in our example.

Also it's obvious that in your case it's possible that the process size will be bigger and the shared memory will be smaller, since you will use different modules and a different code, so you won't get this fantastic ratio, but this example is certainly helps to feel the difference.

1.5.1.5 Preloading Registry Scripts at Server Startup

What happens if you find yourself stuck with Perl CGI scripts and you cannot or don't want to move most of the stuff into modules to benefit from modules preloading, so the code will be shared by the children. Luckily you can preload scripts as well. This time the `Apache::RegistryLoader` modules comes to aid. `Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup.

For example to preload the script `/perl/test.pl` which is in fact the file `/home/httpd/perl/test.pl` you would do the following:

```
use Apache::RegistryLoader ();
Apache::RegistryLoader->new->handler("/perl/test.pl",
    "/home/httpd/perl/test.pl");
```

You should put this code either into `<Perl>` sections or into a startup script.

But what if you have a bunch of scripts located under the same directory and you don't want to list them one by one. Take the benefit of Perl modules and put them to a good use. The `File::Find` module will do most of the work for you.

The following code walks the directory tree under which all `Apache::Registry` scripts are located. For each encountered file with extension `.pl`, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server, before pre-forking the child processes:

```
use File::Find qw(finddepth);
use Apache::RegistryLoader ();
{
    my $scripts_root_dir = "/home/httpd/perl/";
    my $rl = Apache::RegistryLoader->new;
    finddepth
    (
        sub {
            return unless /\.pl$/;
            my $url = "$File::Find::dir/$_";
            $url =~ s|$scripts_root_dir/?||;
            warn "pre-loading $url\n";
            # preload $url
            my $status = $rl->handler($url);
            unless($status == 200) {
                warn "pre-load of '$url' failed, status=$status\n";
            }
        },
        $scripts_root_dir);
}
```

Note that we didn't use the second argument to `handler()` here, as in the first example. To make the loader smarter about the URI to 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()` function was not defined, the loader will try using the URI relative to **ServerRoot**.

A simple `trans()` function can be something like that:

```
sub mytrans {
    my $uri = shift;
    $uri =~ s|^/perl/|/home/httpd/perl/|;
    return $uri;
}
```

You can easily derive the right translation by looking at the `Alias` directive. The above `mytrans()` function is matching our `Alias`:

```
Alias /perl/ /home/httpd/perl/
```

After defining the URI to filename translation function you should pass it during the creation of the `Apache::RegistryLoader` object:

```
my $rl = Apache::RegistryLoader->new(trans => \&mytrans);
```

I won't show any benchmarks here, since the effect is absolutely the same as with preloading modules.

See also `BEGIN` blocks

1.5.1.6 Modules Initializing at Server Startup

We have just learned that it's important to preload the modules and scripts at the server startup. It turns out that it's not enough for some modules and you have to prerun their initialization code to get more memory pages shared. Basically you will find an information about specific modules in their respective manpages. We will present a few examples of widely used modules where the code can be initialized.

1.5.1.6.1 Initializing DBI.pm

The first example is the DBI module. As you know DBI works with many database drivers falling into the `DBD::` category, e.g. `DBD::mysql`. It's not enough to preload DBI, you should initialize DBI with driver(s) that you are going to use (usually a single driver is used), if you want to minimize memory use after forking the child processes. Note that you want to do this under `mod_perl` and other environments where the shared memory is very important. Otherwise you shouldn't initialize drivers.

You probably know already that under `mod_perl` you should use the `Apache::DBI` module to get the connection persistence, unless you open a separate connection for each user--in this case you should not use this module. `Apache::DBI` automatically loads DBI and overrides some of its methods, so you should continue coding like there is only a DBI module.

Just as with modules preloading our goal is to find the startup environment that will lead to the smallest "difference" between the shared and normal memory reported, therefore a smaller total memory usage.

And again in order to have an easy measurement we will use only one child process, therefore we will use this setting in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We always preload these modules:

```
use Gtop();
use Apache::DBI(); # preloads DBI as well
```

We are going to run memory benchmarks on five different versions of the *startup.pl* file.

- **option 1**

Leave the file unmodified.

- **option 2**

Install MySQL driver (we will use MySQL RDBMS for our test):

```
DBI->install_driver("mysql");
```

It's safe to use this method, since just like with `use ()`, if it can't be installed it'll die().

- **option 3**

Preload MySQL driver module:

```
use DBD::mysql;
```

- **option 4**

Tell `Apache::DBI` to connect to the database when the child process starts (`ChildInitHandler`), no driver is preload before the child gets spawned!

```
Apache::DBI->connect_on_init('DBI:mysql:test::localhost',
                             "",
                             "",
                             {
                               PrintError => 1, # warn() on errors
                               RaiseError => 0, # don't die on error
                               AutoCommit => 1, # commit executes
                               # immediately
                             }
                             )
or die "Cannot connect to database: $DBI::errstr";
```

- **option 5**

Options 2 and 4: using `connect_on_init()` and `install_driver()`.

Here is the `Apache::Registry` test script that we have used:

```
preload_dbi.pl
-----
use strict;
use GTop ();
use DBI ();

my $dbh = DBI->connect("DBI:mysql:test::localhost",
                      "",
                      "",
                      {
                        PrintError => 1, # warn() on errors
                        RaiseError => 0, # don't die on error
                        AutoCommit => 1, # commit executes
                                   # immediately
                      }
                      )
    or die "Cannot connect to database: $DBI::errstr";

my $r = shift;
$r->send_http_header('text/plain');

my $do_sql = "show tables";
my $sth = $dbh->prepare($do_sql);
$sth->execute();
my @data = ();
while (my @row = $sth->fetchrow_array){
    push @data, @row;
}
print "Data: @data\n";
$dbh->disconnect(); # NOP under Apache::DBI

my $proc_mem = GTop->new->proc_mem($$);
my $size = $proc_mem->size;
my $share = $proc_mem->share;
my $diff = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Diff);
printf "%8d %8d %8d (bytes)\n", $size, $share, $diff;
```

The script opens a connection to the database `'test'` and issues a query to learn what tables the database has. When the data is collected and printed the connection would be closed in the regular case, but `Apache::DBI` overrides it with empty method. When the data is processed a familiar to you already code to print the memory usage follows.

The server was restarted before each new test.

So here are the results of the five tests that were conducted, sorted by the *Diff* column:

1. After the first request:

Test type	Size	Shared	Diff
install_driver (2)	3465216	2621440	843776
install_driver & connect_on_init (5)	3461120	2609152	851968
preload driver (3)	3465216	2605056	860160
nothing added (1)	3461120	2494464	966656
connect_on_init (4)	3461120	2482176	978944

2. After the second request (all the subsequent request showed the same results):

Test type	Size	Shared	Diff
install_driver (2)	3469312	2609152	860160
install_driver & connect_on_init (5)	3481600	2605056	876544
preload driver (3)	3469312	2588672	880640
nothing added (1)	3477504	2482176	995328
connect_on_init (4)	3481600	2469888	1011712

Now what do we conclude from looking at these numbers. First we see that only after a second reload we get the final memory footprint for a specific request in question (if you pass different arguments the memory usage might and will be different).

But both tables show the same pattern of memory usage. We can clearly see that the real winner is the *startup.pl* file's version where the MySQL driver was installed (2). Since we want to have a connection ready for the first request made to the freshly spawned child process, we generally use the version (5) which uses somewhat more memory, but has almost the same number of shared memory pages. The version (3) only preloads the driver which results in smaller shared memory. The last two versions having nothing initialized (1) and having only the *connect_on_init()* method used (4). The former is a little bit better than the latter, but both significantly worse than the first two versions.

To remind you why do we look for the smallest value in the column *diff*, recall the real memory usage formula:

```
RAM_dedicated_to_mod_perl = diff * number_of_processes
                           + the_processes_with_largest_shared_memory
```

Notice that the smaller the *diff* is, the bigger the number of processes you can have using the same amount of RAM. Therefore every 100K difference counts, when you multiply it by the number of processes. If we take the number from the version (2) vs. (4) and assume that we have 256M of memory dedicated to *mod_perl* processes we will get the following numbers using the formula derived from the above formula:

```

RAM - largest_shared_size
N_of Procs = -----
                Diff

                268435456 - 2609152
(ver 2)  N =  ----- = 309
                860160

                268435456 - 2469888
(ver 4)  N =  ----- = 262
                1011712
```

So you can tell the difference (17% more child processes in the first version).

1.5.1.6.2 *Initializing CGI.pm*

CGI.pm is a big module that by default postpones the compilation of its methods until they are actually needed, thus making it possible to use it under a slow mod_cgi handler without adding a big overhead. That's not what we want under mod_perl and if you use CGI.pm you should precompile the methods that you are going to use at the server startup in addition to preloading the module. Use the compile method for that:

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

where you should replace the tag group :all with the real tags and group tags that you are going to use if you want to optimize the memory usage.

We are going to compare the shared memory foot print by using the script which is back compatible with mod_cgi. You will see that you can improve performance of this kind of scripts as well, but if you really want a fast code think about porting it to use Apache::Request for CGI interface and some other module for HTML generation.

So here is the Apache::Registry script that we are going to use to make the comparison:

```
preload_cgi_pm.pl
-----
use strict;
use CGI ();
use GTop ();

my $q = new CGI;
print $q->header('text/plain');
print join "\n", map {"$_ => ".$q->param($_) } $q->param;
print "\n";

my $proc_mem = GTop->new->proc_mem($$);
my $size = $proc_mem->size;
my $share = $proc_mem->share;
my $diff = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Diff);
printf "%8d %8d %8d (bytes)\n", $size, $share, $diff;
```

The script initializes the CGI object, sends HTTP header and then print all the arguments and values that were passed to the script if at all. At the end as usual we print the memory usage.

As usual we are going to use a single child process, therefore we will use this setting in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We are going to run memory benchmarks on three different versions of the *startup.pl* file. We always preload this module:

```
use Gtop();
```

- **option 1**

Leave the file unmodified.

- **option 2**

Preload CGI.pm:

```
use CGI ();
```

- **option 3**

Preload CGI.pm and pre-compile the methods that we are going to use in the script:

```
use CGI ();
CGI->compile(qw(header param));
```

The server was restarted before each new test.

So here are the results of the five tests that were conducted, sorted by the *Diff* column:

1. After the first request:

Version	Size	Shared	Diff	Test type
1	3321856	2146304	1175552	not preloaded
2	3321856	2326528	995328	preloaded
3	3244032	2465792	778240	preloaded & methods+compiled

2. After the second request (all the subsequent request showed the same results):

Version	Size	Shared	Diff	Test type
1	3325952	2134016	1191936	not preloaded
2	3325952	2314240	1011712	preloaded
3	3248128	2445312	802816	preloaded & methods+compiled

The first version shows the results of the script execution when CGI.pm wasn't preloaded. The second version with module preloaded. The third when it's both preloaded and the methods that are going to be used are precompiled at the server startup.

By looking at the version one of the second table we can conclude that, preloading adds about 20K of shared size. As we have mention at the beginning of this section that's how CGI.pm was implemented--to reduce the load overhead. Which means that preloading CGI is almost hardly change a thing. But if we compare the second and the third versions we will see a very significant difference of 207K (1011712-802816), and we have used only a few methods (the *header* method loads a few more method transparently for a user). Imagine how much memory we are going to save if we are going to precompile

all the methods that we are using in other scripts that use `CGI.pm` and do a little bit more than the script that we have used in the test.

But even in our very simple case using the same formula, what do we see? (assuming that we have 256MB dedicated for `mod_perl`)

$$\text{N_of Procs} = \frac{\text{RAM} - \text{largest_shared_size}}{\text{Diff}}$$

$$\text{(ver 1) } N = \frac{268435456 - 2134016}{1191936} = 223$$

$$\text{(ver 3) } N = \frac{268435456 - 2445312}{802816} = 331$$

If we preload `CGI.pm` and precompile a few methods that we use in the test script, we can have 50% more child processes than when we don't preload and precompile the methods that we are going to use.

META: I've heard that the 3.x generation will be less bloated, so probably I'll have to rerun this using the new version.

1.5.2 Increasing Shared Memory With mergemem

`mergemem` is an experimental utility for linux, which looks *very* interesting for us `mod_perl` users: <http://www.complang.tuwien.ac.at/ulrich/mergemem/>

It looks like it could be run periodically on your server to find and merge duplicate pages. It won't halt your httpds during the merge, this aspect has been taken into consideration already during the design of `mergemem`: Merging is not performed with one big systemcall. Instead most operation is in userspace, making a lot of small systemcalls.

Therefore blocking of the system should not happen. And, if it really should turn out to take too much time you can reduce the priority of the process.

The worst case that can happen is this: `mergemem` merges two pages and immediately afterwards they will be split. The split costs about the same as the time consumed by merging.

This software comes with a utility called `memcmp` to tell you how much you might save.

1.5.3 Forking and Executing Subprocesses from mod_perl

It's desirable to avoid forking under `mod_perl`. Since when you do, you are forking the entire Apache server, lock, stock and barrel. Not only is your Perl code and Perl interpreter being duplicated, but so is `mod_ssl`, `mod_rewrite`, `mod_log`, `mod_proxy`, `mod_speling` (it's not a typo!) or whatever modules you have used in your server, all the core routines, etc.

Modern Operating Systems come with a very light version of fork which adds a little overhead when called, since it was optimized to do the absolute minimum of memory pages duplications. The *copy-on-write* technique is the one that allows to do so. The gist of this technique is as follows: the parent process memory pages aren't immediately copied to the child's space on `fork()`, but this is done only when the child or the parent modifies the data in some memory pages. Before the pages get modified they get marked as dirty and the child has no choice but to copy the pages that are to be modified since they cannot be shared any more.

If you need to call a Perl program from your `mod_perl` code, it's better to try to covert the program into a module and call it a function without spawning a special process to do that. Of course if you cannot do that or the program is not written in Perl, you have to call via `system()` or is equivalent, which spawn a new process. If the program written in C, you may try to write a Perl glue code with help of XS or SWIG architectures, and then the program will be executed as a perl subroutine.

Also by trying to spawn a sub-process, you might be trying to do the *"wrong thing"*. If what you really want is to send information to the browser and then do some post-processing, look into the `Perl-CleanupHandler` directive. The latter allows you to tell the child process after request has been processed and user has received the response. This doesn't release the `mod_perl` process to serve other requests, but it allows to send the response to the client faster. If this is the situation and you need to run some cleanup code, you may want to register this code during the request processing via:

```
my $r = shift;
$r->register_cleanup(\&do_cleanup);
sub do_cleanup{ #some clean-up code here }
```

But when a long term process needs to be spawned, there is not much choice, but to use `fork()`. We cannot just run this long term process within Apache process, since it'll first keep the Apache process busy, instead of letting it do the job it was designed for. And second, if Apache will be stopped the long term process might be terminated as well, unless coded properly to detach from Apache processes group.

In the following sections we are going to discuss how to properly spawn new processes under `mod_perl`.

1.5.3.1 Forking a New Process

This is a typical way to call `fork()` under `mod_perl`:

```
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
} else {
    # Child runs this block
    # some code comes here
    CORE::exit(0);
}
# possibly more code here usually run by the parent
```

When using `fork()`, you should check its return value, since if it returns `undef` it means that the call was unsuccessful and no process was spawned. Something that can happen when the system is running too many processes and cannot spawn new ones.

When the process is successfully forked--the parent receives the PID of the newly spawned child as a returned value of the `fork()` call and the child receives 0. Now the program splits into two. In the above example the code inside the first block after *if* will be executed by the parent and the code inside the first block after *else* will be executed by the child process.

It's important not to forget to explicitly call `exit()` at the end of the child code when forking. Since if you don't and there is some code outside the *if/else block*, the child process will execute it as well. But under `mod_perl` there is another nuance--you must use `CORE::exit()` and not `exit()`, which would be automatically overridden by `Apache::exit()` if used in conjunction with `Apache::Registry` and similar modules. And we want the spawned process to quit when its work is done, otherwise it'll just stay alive use resources and do nothing.

The parent process usually completes its execution path and enters the pool of free servers to wait for a new assignment. If the execution path is to be aborted earlier for some reason one should use `Apache::exit()` or `die()`, in the case of `Apache::Registry` or `Apache::PerlRun` handlers a simple `exit()` will do the right thing.

The child shares with parent its memory pages until it has to modify some of them, which triggers a *copy-on-write* process which copies these pages to the child's domain before the child is allowed to modify them. But this all happens afterwards. At the moment the `fork()` call executed, the only work to be done before the child process goes on its separate way is setting up the page tables for the virtual memory, which imposes almost no delay at all.

1.5.3.2 Freeing the Parent Process

In the child code you must also close all the pipes to the connection socket that were opened by the parent process (i.e. `STDIN` and `STDOUT`) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. If you need the `STDIN` and/or `STDOUT` streams you should re-open them. You may need to close or re-open the `STDERR` filehandle. It's opened to append to the *error_log* file as inherited from its parent, so chances are that you will want to leave it untouched.

Under `mod_perl`, the spawned process also inherits the file descriptor that's tied to the socket through which all the communications between the server and the client happen. Therefore we need to free this stream in the forked process. If we don't do that, the server cannot be restarted while the spawned process is still running. If an attempt is made to restart the server you will get the following error:

```
[Mon Dec 11 19:04:13 2000] [crit]
(98)Address already in use: make_sock:
    could not bind to address 127.0.0.1 port 8000
```

`Apache::SubProcess` comes to help and provides a method `cleanup_for_exec()` which takes care of closing this file descriptor.

So the simplest way to freeing the parent process is to close all three `STD*` streams if we don't need them and untie the Apache socket. In addition you may want to change process' current directory to `/` so the forked process won't keep the mounted partition busy, if this is to be unmounted at a later time. To summarize all this issues, here is an example of the fork that takes care of freeing the parent process.

```

use Apache::SubProcess;
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
} else {
    # Child runs this block
    $r->cleanup_for_exec(); # untie the socket
    chdir '/' or die "Can't chdir to /: $!";
    close STDIN;
    close STDOUT;
    close STDERR;

    # some code comes here

    CORE::exit(0);
}
# possibly more code here usually run by the parent

```

Of course between the freeing the parent code and child process termination the real code is to be placed.

1.5.3.3 Detaching the Forked Process

Now what happens if the forked process is running and we decided that we need to restart the web-server? This forked process will be aborted, since when parent process will die during the restart it'll kill its child processes as well. In order to avoid this we need to detach the process from its parent session, by opening a new session with help of `setsid()` system call, provided by the POSIX module:

```

use POSIX 'setsid';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
} else {
    # Child runs this block
    setsid or die "Can't start a new session: $!";
    ...
}

```

Now the spawned child process has a life of its own, and it doesn't depend on the parent anymore.

1.5.3.4 Avoiding Zombie Processes

Now let's talk about zombie processes.

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

A zombie is a process that doesn't have a parent. When the child quits, it reports the termination to its parent. If no parent `wait()`s to collect the exit status of the child, it gets "*confused*" and becomes a ghost process, that can be seen as a process, but not killed. It will be killed only when you stop the parent process that spawned it!

Generally the `ps(1)` utility displays these processes with the `<defunc>` tag, and you will see the zombies counter increment when doing `top()`. These zombie processes can take up system resources and are generally undesirable.

So the proper way to do a fork is:

```
my $r = shift;
$r->send_http_header('text/plain');

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

In most cases the only reason you would want to fork is when you need to spawn a process that will take a long time to complete. So if the Apache process that spawns this new child process has to wait for it to finish, you have gained nothing. You can neither wait for its completion (because you don't have the time to), nor continue because you will get yet another zombie process. This is called a blocking call, since the process is blocked to do anything else before this call gets completed.

The simplest solution is to ignore your dead children. Just add this line before the `fork()` call:

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

When you set the `CHLD` (`SIGCHLD` in C) signal handler to `'IGNORE'`, all the processes will be collected by the `init` process and are therefore prevented from becoming zombies. This doesn't work everywhere, however. It proved to work at least on Linux OS.

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

[META: Can anyone explain why localization doesn't work?]

So now the code would look like this:

```
my $r = shift;
$r->send_http_header('text/plain');

$SIG{CHLD} = 'IGNORE';

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

Note that `waitpid()` call has gone. The `$SIG{CHLD} = 'IGNORE';` statement protects us from zombies, as explained above.

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

```
my $r = shift;
$r->send_http_header('text/plain');

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
        # do something long lasting
        CORE::exit(0);
    }
}
```

Grandkid becomes a "*child of init*", i.e. the child of the process whose PID is 1.

Note that the previous two solutions do allow you to know the exit status of the process, but in our example we didn't care about it.

Another 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 one 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 for reaping. The handler will loop until `waitpid()` returns a negative number or zero, indicating that no more reappable children remain.

While you test and debug your code that uses one of the above examples, You might want to write some debug information to the `error_log` file so you know what happens.

Read *perlipc* manpage for more information about signal handlers.

1.5.3.5 A Complete Fork Example

Now let's put all the bits of code together and show a well written fork code that solves all the problems discussed so far. We will use an `Apache::Registry` script for this purpose:

```

proper_fork1.pl
-----
use strict;
use POSIX 'setsid';
use Apache::SubProcess;

my $r = shift;
$r->send_http_header("text/plain");

$SIG{CHLD} = 'IGNORE';
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent $$ has finished, kid's PID: $kid\n";
} else {
    $r->cleanup_for_exec(); # untie the socket
    chdir '/' or die "Can't chdir to /: $!";
    open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null'
        or die "Can't write to /dev/null: $!";
    open STDERR, '>/tmp/log' or die "Can't write to /tmp/log: $!";
    setsid or die "Can't start a new session: $!";

    my $oldfh = select STDERR;
    local $| = 1;
    select $oldfh;
    warn "started\n";
    # do something time-consuming
    sleep 1, warn "$_\n" for 1..20;
    warn "completed\n";

    CORE::exit(0); # terminate the process
}

```

The script starts with the usual declaration of the strict mode, loading the POSIX and Apache::SubProcess modules and importing of the setsid() symbol from the POSIX package.

The HTTP header is sent next, with the *Content-type* of *text/plain*. The parent process gets ready to ignore the child, to avoid zombies and the fork is called.

The program gets its personality split after fork and the if conditional evaluates to a true value for the parent process, and to a false value for the child process, therefore the first block is executed by the parent and the second by the child.

The parent process announces his PID and the PID of the spawned process and finishes its block. If there will be any code outside it will be executed by the parent as well.

The child process starts its code by disconnecting from the socket, changing its current directory to /, opening the STDIN and STDOUT streams to /dev/null, which in effect closes them both before opening. In fact in this example we don't need neither of these, so we could just close() both. The child process completes its disengagement from the parent process by opening the STDERR stream to /tmp/log, so it could write there, and creating a new session with help of setsid(). Now the child process has nothing to do with the parent process and can do the actual processing that it has to do. In our example it performs a simple series of warnings, which are logged into /tmp/log:

```

my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";

```

The localized setting of `$| = 1` unbuffers the `STDERR` stream, so we can immediately see the debug output generated by the program. In fact this setting is not required when the output is generated by `warn()`.

Finally the child process terminates by calling:

```
CORE::exit(0);
```

which make sure that it won't get out of the block and run some code that it's not supposed to run.

This code example will allow you to verify that indeed the spawned child process has its own life, and its parent is free as well. Simply issue a request that will run this script, watch that the warnings are started to be written into the `/tmp/log` file and issue a complete server stop and start. If everything is correct, the server will successfully restart and the long term process will still be running. You will know that it's still running, if the warnings will still be printed into the `/tmp/log` file. You may need to raise the number of warnings to do above 20, to make sure that you don't miss the end of the run.

If there are only 5 warnings to be printed, you should see the following output in this file:

```

started
1
2
3
4
5
completed

```

1.5.3.6 Starting a Long Running External Program

But what happens if we cannot just run a Perl code from the spawned process and we have a compiled utility, i.e. a program written in C. Or we have a Perl program which cannot be easily converted into a module, and thus called as a function. Of course in this case we have to use `system()`, `exec()`, `qx()` or `` `` (back ticks) to start it.

When using any of these methods and when the *Taint* mode is enabled, we must at least add the following code to untaint the `PATH` environment variable and delete a few other insecure environment variables. This information can be found in the *perlsec* manpage.

```

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

```

Now all we have to do is to reuse the code from the previous section.

First we move the core program into the *external.pl* file, add the shebang first line so the program will be executed by Perl, tell the program to run under *Taint* mode (-T) and possibly enable the *warnings* mode (-w) and make it executable:

```
external.pl
-----
#!/usr/bin/perl -Tw

open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
open STDOUT, '>/dev/null'
    or die "Can't write to /dev/null: $!";
open STDERR, '>/tmp/log' or die "Can't write to /tmp/log: $!";

my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";
```

Now we replace the code that moved into the external program with `exec()` to call it:

```
proper_fork_exec.pl
-----
use strict;
use POSIX 'setsid';
use Apache::SubProcess;

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

my $r = shift;
$r->send_http_header("text/html");

$SIG{CHLD} = 'IGNORE';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent has finished, kid's PID: $kid\n";
} else {
    $r->cleanup_for_exec(); # untie the socket
    chdir '/' or die "Can't chdir to /: $!";
    open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null'
        or die "Can't write to /dev/null: $!";
    open STDERR, '>&STDOUT' or die "Can't dup stdout: $!";
    setsid or die "Can't start a new session: $!";

    exec "/home/httpd/perl/external.pl" or die "Cannot execute exec: $!";
}
```

Notice that `exec()` never returns unless it fails to start the process. Therefore you shouldn't put any code after `exec()`--it will be not executed in the case of success. Use `system()` or back-ticks instead if you want to continue doing other things in the process. But then you probably will want to terminate the process

after the program has finished. So you will have to write:

```
system "/home/httpd/perl/external.pl" or die "Cannot execute system: $!";
CORE::exit(0);
```

Another important nuance is that we have to close all STD* stream in the forked process, even if the called program does that.

If the external program is written in Perl you may pass complicated data structures to it using one of the methods to serialize Perl data and then to restore it. The *Storable* and *FreezeThaw* modules come handy. Let's say that we have program *master.pl* calling program *slave.pl*:

```
master.pl
-----
# we are within the mod_perl code
use Storable ();
my @params = (foo => 1, bar => 2);
my $params = Storable::freeze(\@params);
exec "./slave.pl", $params or die "Cannot execute exec: $!";

slave.pl
-----
#!/usr/bin/perl -w
use Storable ();
my @params = @ARGV ? @{ Storable::thaw(shift)||[] } : ();
# do something
```

As you can see, *master.pl* serializes the *@params* data structure with *Storable::freeze* and passes it to *slave.pl* as a single argument. *slave.pl* restores the it with *Storable::thaw*, by shifting the first value of the *ARGV* array if available. The *FreezeThaw* module does a very similar thing.

1.5.3.7 Starting a Short Running External Program

Sometimes you need to call an external program and you cannot continue before this program completes its run and optionally returns some result. In this case the fork solution doesn't help. But we have a few ways to execute this program. First using *system()*:

```
system "perl -e 'print 5+5' "
```

We believe that you will never call the perl interpreter for doing this simple calculation, but for the sake of a simple example it's good enough.

The problem with this approach is that we cannot get the results printed to *STDOUT*, and that's where back-ticks or *qx()* come to help. If you use either:

```
my $result = `perl -e 'print 5+5'`;
```

or:

```
my $result = qx{perl -e 'print 5+5'};
```

the whole output of the external program will be stored in the `$result` variable.

Of course you can use other solutions, like opening a pipe (`|` to the program) if you need to submit many arguments and more evolved solutions provided by other Perl modules like `IPC::Open2` which allows to open a process for both reading and writing.

1.5.3.8 Executing `system()` or `exec()` in the Right Way

The `exec()` and `system()` system calls behave identically in the way they spawn a program. For example let's use `system()` as an example. Consider the following code:

```
system("echo", "Hi");
```

Perl will use the first argument as a program to execute, find `/bin/echo` along the search path, invoke it directly and pass the *Hi* string as an argument.

Perl's `system()` is **not** the `system(3)` call [C-library]. This is how the arguments to `system()` get interpreted. When there is a single argument to `system()`, it'll be checked for having shell metacharacters first (like `*`, `?`), and if there are any--Perl interpreter invokes a real shell program (`/bin/sh -c` on Unix platforms). If you pass a list of arguments to `system()`, they will be not checked for metacharacters, but split into words if required and passed directly to the C-level `execvp()` system call, which is more efficient. That's a *very* nice optimization. In other words, only if you do:

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

will the operating system actually `exec()` a copy of `/bin/sh` to parse your command. But even then since `sh` 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 pointing to the existing program code plus your data space, thus will not create this overhead.

1.5.4 OS Specific Parameters for Proxying

Most of the `mod_perl` enabled servers use a proxy front-end server. This is done in order to avoid serving static objects, and also so that generated output which might be received by slow clients does not cause the heavy but very fast `mod_perl` servers from idly waiting.

There are very important OS parameters that you might want to change in order to improve the server performance. This topic is discussed in the section: [Setting the Buffering Limits on Various OSes](#)

1.6 Performance Tuning by Tweaking Apache Configuration

Correct configuration of the `MinSpareServers`, `MaxSpareServers`, `StartServers`, `MaxClients`, and `MaxRequestsPerChild` parameters is very important. There are no defaults. If they are too low, you will under-use the system's capabilities. If they are too high, the chances are 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. With a plain apache server, it's no big deal if you run many servers since the processes are about 1Mb and don't eat a lot of your RAM. Generally the numbers are even smaller with memory sharing. 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: 50x20Mb = 1Gb. Do you have 1Gb of RAM? Maybe not. So how do you tune the parameters? Generally by trying different combinations and benchmarking the server. Again mod_perl processes can be of much smaller size with memory sharing.

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

It is important to make sure that you run the load generator (the client which generates the test requests) on a system that is more powerful than the system being tested. After all we are trying to simulate Internet users, where many users are trying to reach your service at once. Since the number of concurrent users can be quite large, your testing machine must be very powerful and capable of generating a heavy load. Of course you should not run the clients and the server on the same machine. If you do, your test results would be invalid. Clients will eat CPU and memory that should be dedicated to the server, and vice versa.

1.6.1 Configuration Tuning with ApacheBench

We are going to use ApacheBench (ab) utility to tune our server's configuration. We will simulate 10 users concurrently requesting a very light script at `http://www.example.com/perl/access/access.cgi`. Each simulated user makes 10 requests.

```
% ./ab -n 100 -c 10 http://www.example.com/perl/access/access.cgi
```

The results are:

```
Document Path:      /perl/access/access.cgi
Document Length:    16 bytes

Concurrency Level:   10
Time taken for tests: 1.683 seconds
Complete requests:   100
Failed requests:     0
Total transferred:   16100 bytes
HTML transferred:    1600 bytes
Requests per second: 59.42
Transfer rate:       9.57 kb/s received

Connnection Times (ms)
```


	min	avg	max
Connect:	0	29	101
Processing:	77	124	1259
Total:	77	153	1360

The only numbers we really care about are:

Complete requests:	100
Failed requests:	0
Requests per second:	59.42

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

```
% ./ab -n 1000 -c 10 http://www.example.com/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 http://www.example.com/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 133 requests per second! Let's find the upper limit. Using `-n 10000 -c 1000` failed to get results (Broken Pipe?). Using `-n 10000 -c 500` resulted in 94.82 requests per second. 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 each child after it serves 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 http://www.example.com/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 got 8.34 requests per second. 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 reset `MaxRequestsPerChild` to 1500, but reduce `MaxClients` to 10 and run the same test:

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

I got 27.12 requests per second, which is better but still 4-5 times slower. (I got 133 with `MaxClients` set to 50.)

Summary: I have tested a few combinations of the server configuration variables (`MinSpareServers`, `MaxSpareServers`, `StartServers`, `MaxClients` and `MaxRequestsPerChild`). The results I got are as follows:

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

The important parameters are `MaxClients` and `MaxRequestsPerChild`. `MaxClients` should be not too big, so it will not abuse your machine's memory resources, and not too small, for if it is your users will be forced to wait for the children to become free to serve them. `MaxRequestsPerChild` should be as large as possible, to get 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 test script was heavier, the numbers would be different but the conclusions 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 changed the server's configuration, I made sure that the scripts were preloaded by fetching a script at least once for every child.

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

Now let's take a look at some real code whose execution time is more than a few milliseconds. We will do some real testing and collect the data into 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 limited) (http://www.example.com/perl/access/access.cgi?do_sub=query_form), with the configuration:

```
MinSpareServers    8
MaxSpareServers    16
StartServers       10
MaxClients         50
MaxRequestsPerChild 5000
```

gives us:

NR	NC	RPS	comment
10	10	3.33	# not a reliable figure
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 limited by some external operation) 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 the ability to queue the clients, but be aware that anything that goes into the queue means a waiting client and a client (browser) that might time out!

Now we will benchmark the same script without using the mysql (code limited by perl only): (<http://www.example.com/perl/access/access.cgi>), it's the same script but it just returns the HTML form, without making SQL queries.

```
MinSpareServers    8
MaxSpareServers    16
StartServers       10
MaxClients         50
MaxRequestsPerChild 5000
```

NR	NC	RPS	comment
10	10	26.95	# not a reliable figure
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 limited by I/O or mysql), so we see that the server serves the requests much faster. You can see the number of requests per second is almost the same for any load, but goes lower when the number of concurrent clients goes beyond `MaxClients`. With 25 RPS, the machine simulating a load of 400 concurrent clients will be served in 16 seconds. To be more realistic, assuming a maximum of 100 concurrent clients and 30 requests per second, the client will be served in 3.5 seconds. Pretty good for a highly loaded server.

Now we will use the server to its full capacity, by keeping all `MaxClients` clients alive all the time and having a big `MaxRequestsPerChild`, so that no child 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 change `MaxClients` and watch the results: Let's reduce `MaxClients` to 10.

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

NR	NC	RPS	comment

10	10	23.87	# not a reliable figure
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: Very little difference! Ten servers were able to serve almost with the same throughput as 50 servers. Why? My guess is because of CPU throttling. It seems that 10 servers were serving requests 5 times faster than when we worked with 50 servers. In that case, each child received its CPU time slice five 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 drastically to reduce 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 MaxRequestsPerChild, the performance starts to become closer to plain mod_cgi.

Here are the numbers of this run with mod_cgi, for comparison:

```
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 the first test, when NR/NC was 100/10, mod_cgi was capable of 1.12 requests per second. In the same circumstances, mod_perl was capable of 32 requests per second, nearly 30 times faster! In the first test each client waited about 100 seconds to be served. In the second and third tests they waited 1000 seconds!

1.6.2 Choosing MaxClients

The MaxClients directive sets the limit on the number of simultaneous requests that can be supported. No more than this number of child server processes will be created. To configure more than 256 clients, you must edit the HARD_SERVER_LIMIT entry in httpd.h and recompile. In our case we want this variable to be as small as possible, because in this way we can limit the resources used by the server children. Since we can restrict each child's process size (see Preventing Your Processes from Growing), the calculation of MaxClients is pretty straightforward:

```

                                Total RAM Dedicated to the Webserver
MaxClients = -----
                                MAX child's process size
```

So if I have 400Mb left for the webserver to run with, I can set `MaxClients` to be of 40 if I know that each child is limited to 10Mb of memory (e.g. with `Apache::SizeLimit`).

You will be wondering what will happen to your server if there are more concurrent users than `MaxClients` at any time. This situation is signified by the following warning message in the `error_log`:

```
[Sun Jan 24 12:05:32 1999] [error] server reached MaxClients setting,
consider raising the MaxClients setting
```

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

It **is an error** because clients are being put in the queue rather than getting served immediately, 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 child processes. The best approach is to try not to have this condition reached at all, and if you 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 memory between them when the OS supports that. You must take action to allow the sharing to happen - See `Preload Perl` modules at server startup. If you do this, the 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 a solution please let us know! If the shared memory was of the same size throughout the child's life, we could derive a much better formula:

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

which is:

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

Let's roll some calculations:

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

$$\text{MaxClients} = \frac{500 - 4}{10 - 4} = 82$$

With no sharing in place

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

With sharing in place you can have 64% more servers without buying more RAM.

If you improve sharing and keep the sharing level, let's say:

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

$$\text{MaxClients} = \frac{500 - 8}{10 - 8} = 246$$

392% more servers! Now you can feel the importance of having as much shared memory as possible.

1.6.3 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 solves some memory leakage problems caused by sloppy programming practices, whereas a child process consumes 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. This will allow the system to reclaim the memory that a greedy child process consumed, when it exits after `MaxRequestsPerChild` requests.

But beware -- if you set this number too low, you will lose some of the speed bonus you get from `mod_perl`. Consider using `Apache::PerlRun` if this is the case.

Another approach is to use the `Apache::SizeLimit` or `Apache::GTopLimit` modules. By using either of these modules you should be able to discontinue using the `MaxRequestPerChild`, although for some developers, using both in combination does the job. In addition these modules allow you to kill httpd processes whose shared memory size drops below a specified limit or unshared memory size crosses a specified threshold.

See also Preload Perl modules at server startup and Sharing Memory.

1.6.4 Choosing MinSpareServers, MaxSpareServers and StartServers

With `mod_perl` enabled, it might take as much as 20 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. 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 handle new requests immediately. 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 (assuming 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 shouldn't be used as a rule, but rather as examples to show you some possible scenarios. Use this information with caution!

1.6.5 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` or `Apache::GTopLimit`, you can set this parameter to 0 (treated as infinity).

- **StartServers**

If you keep a small number of servers active most of the time, keep this number low. Keep it low especially if `MaxSpareServers` is also low, as if there is no load Apache will kill its children before they have been utilized at all. If your service is heavily loaded, make this number close to `MaxClients`, and keep `MaxSpareServers` equal to `MaxClients`.

- **MinSpareServers**

If your server performs other work besides web serving, make this low so the memory of unused children will be freed when the load is light. 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 are waiting to handle bursts of requests.

- **MaxSpareServers**

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

- **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 very long). However, do not set it too high. With a high `MaxClients`, if you get a high load the server will try to serve all requests immediately. Your CPU will have a hard time keeping up, and if the child size * number of running children is larger than the total available RAM your server will start swapping. This will slow down everything, which in turn will make things even 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 routinely. If you are low on memory and you badly need it, buy it. Memory is cheap.

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 improvement in performance. 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 region on the curve of the graph of latency and/or throughput against `MaxClients` where the improvement starts to diminish. Stop there and use it. When you make the measurements on a production server you will have the ability to tune them more precisely, since you will see the real numbers.

Don't forget that if you add more scripts, or even just modify the existing ones, the processes will grow in size as you compile in more code. Probably the parameters will need to be recalculated.

1.6.6 KeepAlive

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

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

you have a real performance penalty, since after completing the processing for each request, the process will wait for `KeepAliveTimeout` seconds before closing the connection and will therefore not be serving other requests during this time. With this configuration you will need many more concurrent 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.

The chances are that you don't want this feature enabled. Set it Off with:

```
KeepAlive Off
```

the other two directives don't matter if `KeepAlive` is Off.

You might want to consider enabling this option if the client's browser needs to request more than one object from your server for a single HTML page. If this is the situation then by setting `KeepAlive On` then for each page you save the HTTP 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 serves them all during a single connection. However, your client will see a slightly slower response, since banners will be brought one at a time and not concurrently as is the case if each `IMG` tag opens a separate connection.

Since keepalive connections will not incur the additional three-way TCP handshake they are kinder to the network.

SSL connections benefit the most from `KeepAlive` in case you didn't configure the server to cache session ids.

You have probably followed the advice to send all the requests for static objects to a plain Apache server. Since most pages include more than one unique static image, you should keep the default `KeepAlive` setting of the non-mod_perl server, i.e. keep it On. It will probably be a good idea also to reduce the timeout a little.

One option 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. Obviously you would make new connections to the server as required by the client's requests.

1.6.7 *PerlSetupEnv Off*

`PerlSetupEnv Off` is another optimization you might consider. This directive requires `mod_perl` 1.25 or later.

When this option is enabled, *mod_perl* fiddles with the environment to make it appear as if the code is called under the `mod_cgi` handler. For example, the `$ENV{QUERY_STRING}` environment variable is initialized with the contents of *Apache::args()*, and the value returned by *Apache::server_hostname()* is put into `$ENV{SERVER_NAME}`.

But `%ENV` population is expensive. Those who have moved to the Perl Apache API no longer need this extra `%ENV` population, and can gain by turning it Off. Scripts using the `CGI.pm` module require `PerlSetupEnv On` because that module relies on a properly populated CGI environment table.

By default it is turned On.

Note that you can still set environment variables when `PerlSetupEnv` is turned Off. For example when you use the following configuration:

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

and you issue a request for this script:

```
setupenvoff.pl
-----
use Data::Dumper;
my $r = Apache->request();
$r->send_http_header('text/plain');
print Dumper(\%ENV);
```

you should see something like this:

```
$VAR1 = {
    'GATEWAY_INTERFACE' => 'CGI-Perl/1.1',
    'MOD_PERL' => 'mod_perl/1.25',
    'PATH' => '/usr/lib/perl5/5.00503:... snipped ...',
    'TEST' => 'hi'
};
```

Note that we got the value of the *TEST* environment variable we set in *httpd.conf*.

1.6.8 Reducing the Number of stat() Calls Made by Apache

If you watch the system calls that your server makes (using *truss* or *strace* while processing a request, you will notice that a few stat() calls are made. For example when I fetch `http://localhost/perl-status` and I have my DocRoot set to `/home/httpd/docs` I see:

```
[snip]
stat("/home/httpd/docs/perl-status", 0xbffff8cc) = -1
                                ENOENT (No such file or directory)
stat("/home/httpd/docs", {st_mode=S_IFDIR|0755,
                                st_size=1024, ...}) = 0
[snip]
```

If you have some dynamic content and your virtual relative URI is something like `/news/perl/mod_perl/summary` (i.e., there is no such directory on the web server, the path components are only used for requesting a specific report), this will generate five(!) stat() calls, before the Document-Root is found. You will see something like this:

```
stat("/home/httpd/docs/news/perl/mod_perl/summary", 0xbffff744) = -1
                                ENOENT (No such file or directory)
stat("/home/httpd/docs/news/perl/mod_perl", 0xbffff744) = -1
                                ENOENT (No such file or directory)
stat("/home/httpd/docs/news/perl", 0xbffff744) = -1
                                ENOENT (No such file or directory)
stat("/home/httpd/docs/news", 0xbffff744) = -1
                                ENOENT (No such file or directory)
stat("/home/httpd/docs",
    {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
```

How expensive those calls are? Let's use the `Time::HiRes` module to find out.

```
stat_call_sample.pl
-----
use Time::HiRes qw(gettimeofday tv_interval);
my $calls = 1_000_000;

my $start_time = [ gettimeofday ];

stat "/foo" for 1..$calls;

my $end_time = [ gettimeofday ];

my $elapsed = tv_interval($start_time,$end_time) / $calls;

print "The average execution time: $elapsed seconds\n";
```

This script takes a time sample at the beginnig, then does 1_000_000 stat() calls to a non-existing file, samples the time at the end and prints the average time it took to make a single stat() call. I'm sampling a 1M stats, so I'd get a correct average result.

Before we actually run the script one should distinguish between two different situation. When the server is idle the time between the first and the last system call will be much shorter than the same time measured on the loaded system. That is because on the idle system, a process can use CPU very often, and on the

loaded system lots of processes compete over it and each process has to wait for a longer time to get the same amount of CPU time.

So first we run the above code on the unloaded system:

```
% perl stat_call_sample.pl
The average execution time: 4.209645e-06 seconds
```

So it takes about 4 microseconds to execute a stat() call. Now let start a CPU intensive process in one console. The following code keeps CPU busy all the time.

```
% perl -e '1**1 while 1'
```

And now run the *stat_call_sample.pl* script in the other console.

```
% perl stat_call_sample.pl
The average execution time: 8.777301e-06 seconds
```

You can see that the average time has doubled (about 8 microseconds). And this is obvious, since there were two processes competing over CPU. Now if run 4 occurrences of the above code:

```
% perl -e '1**1 while 1' &
% perl -e '1**1 while 1' &
% perl -e '1**1 while 1' &
% perl -e '1**1 while 1' &
```

And when running our script in parallel with these processes, we get:

```
% perl stat_call_sample.pl
2.0853558e-05 seconds
```

about 20 microseconds. So the average stat() system call is 5 times longer now. Now if you have 50 mod_perl processes that keep the CPU busy all the time, the stat() call will be 50 times slower and it'll take 0.2 milliseconds to complete a series of call. If you have five redundant calls as in the strace example above, they adds up to one millisecond. If you have more processes constantly consuming CPU, this time adds up. Now multiply this time by the number of processes that you have and you get a few seconds lost. As usual, for some services this loss is insignificant, while for others a very significant one.

So why Apache does all these redundant stat() calls? You can blame the default installed TransHandler for this inefficiency. Of course you could supply your own, which will be smart enough not to look for this virtual path and immediately return OK. But in cases where you have a virtual host that serves only dynamically generated documents, you can override the default PerlTransHandler with this one:

```
PerlModule Apache::Constants
<VirtualHost 10.10.10.10:80>
...
PerlTransHandler Apache::Constants::OK
...
</VirtualHost>
```

As you see it affects only this specific virtual host.

This has the effect of short circuiting the normal `TransHandler` processing of trying to find a filesystem component that matches the given URI -- no more 'stat's!

Watching your server under `strace/truss` can often reveal more performance hits than trying to optimize the code itself!

For example unless configured correctly, Apache might look for the `.htaccess` file in many places, if you don't have one and add many `open()` calls.

Let's start with this simple configuration, and will try to reduce the number of irrelevant system calls.

```
DocumentRoot "/home/httpd/docs"
<Location /foo/test>
    SetHandler perl-script
    PerlHandler Apache::Foo
</Location>
```

The above configuration allows us to make a request to `/foo/test` and the Perl handler() defined in `Apache::Foo` will be executed. Notice that in the test setup there is no file to be executed (like in `Apache::Registry`). There is no `.htaccess` file as well.

This is a typical generated trace.

```
stat("/home/httpd/docs/foo/test", 0xbffff8fc) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs/foo", 0xbffff8fc) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs",
    {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
open("/.htaccess", O_RDONLY) = -1 ENOENT
(No such file or directory)
open("/home/.htaccess", O_RDONLY) = -1 ENOENT
(No such file or directory)
open("/home/httpd/.htaccess", O_RDONLY) = -1 ENOENT
(No such file or directory)
open("/home/httpd/docs/.htaccess", O_RDONLY) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs/test", 0xbffff774) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs",
    {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
```

Now we modify the `<Directory>` entry and add `AllowOverride None`, which among other things disables `.htaccess` files and will not try to open them.

```
<Directory />
    AllowOverride None
</Directory>
```

We see that the four open() calls for *.htaccess* have gone.

```
stat("/home/httpd/docs/foo/test", 0xbffff8fc) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs/foo", 0xbffff8fc) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs",
{st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
stat("/home/httpd/docs/test", 0xbffff774) = -1 ENOENT
(No such file or directory)
stat("/home/httpd/docs",
{st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
```

Let's try to shortcut the *foo* location with:

```
Alias /foo /
```

Which makes Apache to look for the file in the */* directory and not under */home/httpd/docs/foo*. Let's run it:

```
stat("//test", 0xbffff8fc) = -1 ENOENT (No such file or directory)
```

Wow, we've got only one stat call left!

Let's remove the last *Alias* setting and use:

```
PerlModule Apache::Constants
PerlTransHandler Apache::Constants::OK
```

as explained above. When we issue the request, we see no stat() calls. But this is possible only if you serve only dynamically generated documents, i.e. no CGI scripts. Otherwise you will have to write your own *PerlTransHandler* to handle requests as desired.

For example this *PerlTransHandler* will not lookup the file on the filesystem if the URI starts with */foo*, but will use the default *PerlTransHandler* otherwise:

```
PerlTransHandler 'sub { return shift->uri() =~ m|^/foo| \
? Apache::Constants::OK \
: Apache::Constants::DECLINED; }'
```

Let's see the same configuration using the *<Perl>* section and a dedicated package:

```
<Perl>
package My::Trans;
use Apache::Constants qw(:common);
sub handler{
    my $r = shift;
    return OK if $r->uri() =~ m|^/foo|;
    return DECLINED;
}
```

```
package Apache::ReadConfig;
$PerlTransHandler = "My::Trans";
</Perl>
```

As you see we have defined the `My::Trans` package and implemented the `handler()` function. Then we have assigned this handler to the `PerlTransHandler`.

Of course you can move the code in the module into an external file, (e.g. *My/Trans.pm*) and configure the `PerlTransHandler` with

```
PerlTransHandler My::Trans
```

in the normal way (no `<Perl>` section required).

There is an even simpler way to save that last `stat()` call. Instead of using `PerlTransHandler` combined with:

```
Alias /foo /
```

we can use:

```
AliasMatch ^/foo /
```

which in the current implementation (at least in apache-1.3.28) doesn't incur the `stat()` call. Using the regex instead of prefix matching might slow things a bit, but is probably still faster than the `stat()` call.

1.7 TMTOWTDI: Convenience and Habit vs. Performance

TMTOWTDI (sometimes pronounced "*tim toady*"), or "*There's More Than One Way To Do It*" is the main motto of Perl. In other words, you can gain the same goal by coding in many different styles, using different modules and deploying the same modules in different ways.

Unfortunately when you come to the point where performance is the goal, you might have to learn what's more efficient and what's not. Of course it might mean that you will have to use something that you don't really like, it might be less convenient or it might be just a matter of habit that one should change.

So this section is about performance trade-offs. For almost each comparison we will provide the theoretical difference and then run benchmarks to support the theory, since however good the theory its the numbers we get in practice that matter.

"Premature optimizations are evil", the saying goes. I believe that knowing how to write an efficient code in first place, where it doesn't make the quality and clarity suffer saves time in the long run. That's what this section is mostly about.

In the following benchmarks, unless told different the following Apache configuration has been used:


```
MinSpareServers 10
MaxSpareServers 20
StartServers 10
MaxClients 20
MaxRequestsPerChild 10000
```

1.7.1 Apache::Registry PerlHandler vs. Custom PerlHandler

At some point you have to decide whether to use `Apache::Registry` and similar handlers and stick to writing scripts for the content generation or to write pure Perl handlers.

`Apache::Registry` maps a request to a file and generates a subroutine to run the code contained in that file. If you use a PerlHandler `My::Handler` instead of `Apache::Registry`, you have a direct mapping from request to subroutine, without the steps in between. These steps include:

1. run the `stat()` on the script's filename (`$r->filename`)
2. check that the file exists and is executable
3. generate a Perl package name based on the request's URI (`$r->uri`)
4. go to the directory the script resides in (`chdir basename $r->filename`)
5. compare the file's and stored in memory compiled subroutine's last modified time (if it was compiled already)
6. if modified or not compiled, compile the subroutine
7. go back to the previous directory (`chdir $old_cwd`)

If you cut out those steps, you cut out some overhead, plain and simple. Do you *need* to cut out that overhead? May be yes, may be not. Your requirements determine that.

You should take a look at the sister `Apache::Registry` modules (e.g. `Apache::RegistryNG` and `Apache::RegistryBB`) that don't perform all these steps, so you can still choose to stick to using scripts to generate the content. The greatest added value of scripts is that you don't have to modify the configuration file to add the handler configuration and restarting the server for each newly written content handler.

Now let's run benchmarks and compare.

We want to see the overhead that `Apache::Registry` adds compared to the custom handler and whether it becomes insignificant when used for the heavy and time consuming code. In order to do that we will run two benchmarks sets: the first so called a *light* set will use an almost empty script, that only sends a basic header and one word as content; the second will be a *heavy* set which will add some time consuming operation to the script's and the handler's code.

For the *light* set we are going to use the *registry.pl* script running under `Apache::Registry`:

```
benchmarks/registry.pl
-----
use strict;
print "Content-type: text/plain\r\n\r\n";
print "Hello";
```

And the following content generation handler:

```
Benchmark/Handler.pm
-----
package Benchmark::Handler;
use Apache::Constants qw(:common);

sub handler{
    $r = shift;
    $r->send_http_header('text/html');
    $r->print("Hello");
    return OK;
}
1;
```

We will add this settings to *httpd.conf*:

```
PerlModule Benchmark::Handler
<Location /benchmark_handler>
    SetHandler perl-script
    PerlHandler Benchmark::Handler
</Location>
```

The first directive worries to preload and compile the `Benchmark::Handler` module. The rest of the lines tell Apache to execute the subroutine `Benchmark::Handler::handler` when a request with relative URI */benchmark_handler* is made.

We will use the usual configuration for `Apache::Registry` scripts, where all the URIs starting with */perl* are remapped to the files residing under */home/httpd/perl/* directory.

```
Alias /perl/ /home/httpd/perl/
<Location /perl>
    SetHandler perl-script
    PerlHandler +Apache::Registry
    Options ExecCGI
    PerlSendHeader On
</Location>
```

We will use the `Apache::RegistryLoader` to preload and compile the script at the server startup as well, so the benchmark will be fair through the benchmark and only the processing time will be measured. To accomplish the preloading we add the following code to the *startup.pl* file:

```
use Apache::RegistryLoader ();
Apache::RegistryLoader->new->handler(
    "/perl/benchmarks/registry.pl",
    "/home/httpd/perl/benchmarks/registry.pl");
```

To create the *heavy* benchmark set let's leave the above code examples unmodified but add some CPU intensive processing operation (it can be also an IO operation or a database query.)

```
my $x = 100;
my $y = log ($x ** 100) for (0..10000);
```

This code does lots of mathematical processing and therefore very CPU intensive.

Now we are ready to proceed with the benchmark. We will generate 5000 requests with 15 as a concurrency level using the `Apache::Benchmark` module.

Here are the reported results:

name	avtime	rps
light handler	15	911
light registry	21	680
heavy handler	183	81
heavy registry	191	77

Let's look at the results and answer the previously asked questions.

First let's compare the results from the *light* set. We can see that the average overhead added by `Apache::Registry` (compared to the custom handler) is about:

```
21 - 15 = 6 milliseconds
```

per request.

Thus the difference in speed is about 40% (15 vs. 21). Note that this doesn't mean that the difference in the real world applications is such big. And the results of the *heavy* set confirm that.

In the *heavy* set the average processing time is almost the same for the `Apache::Registry` and the custom handler. You can clearly see that the difference between the two is almost the same one that we have seen in the *light* set's results. It has grown from 6 milliseconds to 8 milliseconds (191-183). Which means that the identical heavy code that has been added was running for about 168 milliseconds (183-15). It doesn't mean that the added code itself has been running for 168 milliseconds. It means that it took 168 milliseconds for this code to be completed in a multi-process environment where each process gets a time slice to use the CPU. The more processes are running the more time the process will have to wait to get the next time slice when it can use the CPU.

We have the second question answered as well. You can see that when the code is not just the *hello* script, the overhead of the extra operations done but the `Apache::Registry` module, is almost insignificant. It's a non zero though, so it depends on your requirements, and if another 5-10 milliseconds overhead are quite tolerable, you may choose to use `Apache::Registry`.

The interesting thing is that when the server under test runs on a very slow machine the results are completely different. I'll present them here for comparison:

name	avtime	rps
light handler	50	196
light registry	160	61
heavy handler	149	67
heavy registry	822	12

First of all the difference of 6 milliseconds in the average processing time we have seen on the fast machine when running the *light* set, now has grown to 110 milliseconds. Which means that a few extra operations, that `Apache::Registry` does, turn to be very expensive on the slow machine.

Second, you can see that when the *heavy* set is used, there is no preservation of the 110 milliseconds as we have seen on the fast machine, which we obviously would expect to see, since the code that was added should take the same time to execute in the handler and the script. But instead we see a difference of 673 milliseconds (822-149).

The explanation lies in fact that the difference between the machines isn't merely in the CPU speed. It's possible that there are many other things that are different. For example the size of the processor cache. If one machine has a processor cache large enough to hold the whole handler and the other doesn't this can be very significant, given that in our *heavy* benchmark set, 99.9% of the CPU activity was dedicated to running the calculation code.

But this also shows you again, that none of the results and conclusion made here should be taken for granted. Certainly, most chances are that you will see a similar behavior on your machine, but only after you have run the benchmarks and analyzed the received results, you can be sure what is the best for you using the setup under test. If you later you happen to use a different machine, make sure to run the tests again, as they can lead to complete different decision as we have just seen when we have tried the same benchmark on a different machine.

1.7.2 "Bloatware" modules

Perl modules like `IO::` are very convenient, but let's see what it costs us to use them. (perl5.6.0 over OpenBSD)

```
% wc `perl -MIO -e 'print join("\n", sort values %INC, "")'`
124      696    4166 /usr/local/lib/perl5/5.6.0/Carp.pm
580     2465   17661 /usr/local/lib/perl5/5.6.0/Class/Struct.pm
400     1495   10455 /usr/local/lib/perl5/5.6.0/Cwd.pm
313     1589   10377 /usr/local/lib/perl5/5.6.0/Exporter.pm
225      784    5651 /usr/local/lib/perl5/5.6.0/Exporter/Heavy.pm
 92      339    2813 /usr/local/lib/perl5/5.6.0/File/Spec.pm
442     1574   10276 /usr/local/lib/perl5/5.6.0/File/Spec/Unix.pm
115      398    2806 /usr/local/lib/perl5/5.6.0/File/stat.pm
406     1350   10265 /usr/local/lib/perl5/5.6.0/IO/Socket/INET.pm
143      429    3075 /usr/local/lib/perl5/5.6.0/IO/Socket/UNIX.pm
```

```

7168 24137 178650 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Config.pm
230 1052 5995 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Errno.pm
222 725 5216 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Fcntl.pm
47 101 669 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO.pm
239 769 5005 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Dir.pm
169 549 3956 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/File.pm
594 2180 14772 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Handle.pm
252 755 5375 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Pipe.pm
77 235 1709 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Seekable.pm
428 1419 10219 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Socket.pm
452 1401 10554 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Socket.pm
127 473 3554 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/XSLoader.pm
52 161 1050 /usr/local/lib/perl5/5.6.0/SelectSaver.pm
139 541 3754 /usr/local/lib/perl5/5.6.0/Symbol.pm
161 609 4081 /usr/local/lib/perl5/5.6.0/Tie/Hash.pm
109 390 2479 /usr/local/lib/perl5/5.6.0/strict.pm
79 370 2589 /usr/local/lib/perl5/5.6.0/vars.pm
318 1124 11975 /usr/local/lib/perl5/5.6.0/warnings.pm
30 85 722 /usr/local/lib/perl5/5.6.0/warnings/register.pm
13733 48195 349869 total

```

Moreover, that requires 116 happy trips through the kernel's `namei()`. It syscalls `open()` a remarkable 57 times, 17 of which failed but leaving 38 that were successful. It also syscalled `read()` a curiously identical 57 times, ingesting a total of 180,265 plump bytes. To top it off, this *increases your resident set size by two megabytes!*

Happy mallocing...

It seems that `CGI.pm` suffers from the same disease:

```

% wc `perl -MCGI -le 'print for values %INC'`
1368 6920 43710 /usr/local/lib/perl5/5.6.0/overload.pm
6481 26122 200840 /usr/local/lib/perl5/5.6.0/CGI.pm
7849 33042 244550 total

```

You have 16 trips through `namei`, 7 successful opens, 2 unsuccessful ones, and 213k of data read in.

This is a *perlbloat.pl* that shows how much memory is acquired by Perl when you run some. So we can easily test the overhead of loading some modules.

```

#!/usr/bin/perl -w

use GTop ();

my $gtop = GTop->new;
my $before = $gtop->proc_mem($$)->size;

for (@ARGV) {
    if (eval "require $_") {
        eval {
            $_->import;
        };
    }
    else {
        eval $_;
    }
}

```

1.7.3 Apache::args vs. Apache::Request::param vs. CGI::param

```
        die $@ if $@;
    }
}

my $after = $gtop->proc_mem($$)->size;
printf "@ARGV added %s\n", GTop::size_string($after - $before);
```

Now let's try to load IO, which loads IO::Handle, IO::Seekable, IO::File, IO::Pipe, IO::Socket and IO::Dir:

```
% ./perlbloat.pl 'use IO;'
use IO; added 1.5M
```

"Only" 1.5 MB overhead. Now let's load CGI (v2.74) and compile all its methods:

```
% ./perlbloat.pl 'use CGI; CGI->compile(":all")'
use CGI; CGI->compile(":all") added 1.8M
```

Almost 2MB extra memory. Let's compare CGI.pm with its younger sister, whose internals are implemented in C.

```
%. /perlbloat.pl 'use Apache::Request'
use Apache::Request added 48k
```

48KB. A significant difference isn't it?

The following numbers show memory sizes in KB (virtual and resident) for v5.6.0 of Perl on four different operating systems, The three calls each are without any modules, with just -MCGI, and with -MIO (never with both):

	OpenBSD		FreeBSD		Redhat Linux		Solaris	
	vsz	rss	vsz	rss	vsz	rss	vsz	rss
Raw Perl	736	772	832	1208	2412	980	2928	2272
w/ CGI	1220	1464	1308	1828	2972	1768	3616	3232
w/ IO	2292	2580	2456	3016	4080	2868	5384	4976

Anybody who's thinking of choosing one of these might do well to digest these numbers first.

1.7.3 Apache::args vs. Apache::Request::param vs. CGI::param

Apache::args, Apache::Request::param and CGI::param are the three most common ways to process input arguments in mod_perl handlers and scripts. Let's write three Apache::Registry scripts that use Apache::args, Apache::Request::param and CGI::param to process a form's input and print it out. Notice that Apache::args is considered identical to Apache::Request::param only when you have single valued keys. In the case of multi-valued keys (e.g. when using check-box groups) you will have to write some extra code: If you do a simple:

```
my %params = $r->args;
```

only the last value will be stored and the rest will collapse, because that's what happens when you turn a list into a hash. Assuming that you have the following list:

```
(rules => 'Apache', rules => 'Perl', rules => 'mod_perl')
```

and assign it to a hash, the following happens:

```
$hash{rules} = 'Apache';
$hash{rules} = 'Perl';
$hash{rules} = 'mod_perl';
```

So at the end only the:

```
rules => 'mod_perl'
```

pair will get stored. With CGI.pm or Apache::Request you can solve this by extracting the whole list by its key:

```
my @values = $q->params('rules');
```

In addition Apache::Request and CGI.pm have many more functions that ease input processing, like handling file uploads. However Apache::Request is much faster since its guts are implemented in C, glued to Perl using XS code.

Assuming that the only functionality you need is the parsing of key-value pairs, and assuming that every key has a single value, we will compare the following almost identical scripts, by trying to pass various query strings.

Here's the code:

```
file:processing_with_apache_args.pl
-----
use strict;
my $r = shift;
$r->send_http_header('text/plain');
my %args = $r->args;
print join "\n", map {"$_ => ".$args{$_} } keys %args;

file:processing_with_apache_request.pl
-----
use strict;
use Apache::Request ();
my $r = shift;
my $q = Apache::Request->new($r);
$r->send_http_header('text/plain');
my %args = map {"$_ => $q->param($_) } $q->param;
print join "\n", map {"$_ => ".$args{$_} } keys %args;
```

1.7.3 Apache::args vs. Apache::Request::param vs. CGI::param

```
file:processing_with_cgi_pm.pl
-----
use strict;
use CGI;
my $r = shift;
$r->send_http_header('text/plain');
my $q = new CGI;
my %args = map {$_ => $q->param($_)} $q->param;
print join "\n", map {"$_ => ".$args{$_}} keys %args;
```

All three scripts are preloaded at server startup:

```
<Perl>
  use Apache::RegistryLoader ();
  Apache::RegistryLoader->new->handler(
    "/perl/processing_with_cgi_pm.pl",
    "/home/httpd/perl/processing_with_cgi_pm.pl"
  );
  Apache::RegistryLoader->new->handler(
    "/perl/processing_with_apache_request.pl",
    "/home/httpd/perl/processing_with_apache_request.pl"
  );
  Apache::RegistryLoader->new->handler(
    "/perl/processing_with_apache_args.pl",
    "/home/httpd/perl/processing_with_apache_args.pl"
  );
</Perl>
```

We use four different query strings, generated by:

```
my @queries = (
  join("&", map {"$_=" . 'e' x 10} ('a'..'b')),
  join("&", map {"$_=" . 'e' x 50} ('a'..'b')),
  join("&", map {"$_=" . 'e' x 5} ('a'..'z')),
  join("&", map {"$_=" . 'e' x 10} ('a'..'z')),
);
```

The first string is:

```
a=eeeeeeeeee&b=eeeeeeeeee
```

which is 25 characters in length and consists of two key/value pairs. The second string is also made of two key/value pairs, but the value is 50 characters long (total 105 characters). The third and the forth strings are made from 26 key/value pairs, with the value lengths of 5 and 10 characters respectively, with total lengths of 207 and 337 characters respectively. The `query_len` column in the report table is one of these four total lengths.

We conduct the benchmark with concurrency level of 50 and generate 5000 requests for each test.

And the results are:

```
-----
name    val_len pairs query_len | avtime  rps
-----
apreq    10      2      25   |    51   945
```


apreq	50	2	105		53	907
r_args	50	2	105		53	906
r_args	10	2	25		53	899
apreq	5	26	207		64	754
apreq	10	26	337		65	742
r_args	5	26	207		73	665
r_args	10	26	337		74	657
cgi_pm	50	2	105		85	573
cgi_pm	10	2	25		87	559
cgi_pm	5	26	207		188	263
cgi_pm	10	26	337		188	262

Where `apreq` stands for `Apache::Request::param()`, `r_args` stands for `Apache::args()` or `$r->args()` and `cgi_pm` stands for `CGI::param()`.

You can see that `Apache::Request::param` and `Apache::args` have similar performance with a few key/value pairs, but the former is faster with many key/value pairs. `CGI::param` is significantly slower than the other two methods.

1.7.4 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 multiple `print()` calls (`_bad_` style in generating output) or if you just have too many of them, then you will experience a degradation in performance. The severity depends on the number of `print()` calls that you make.

Many old CGI scripts were written like this:

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

This example has multiple `print()` calls, which will cause performance degradation with `$|=1`. It also uses too many backslashes. This makes the code less readable, and it is also more difficult to format the HTML so that it is easily readable as the script's output. The code below solves the problems:

```
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 expect the first characters after the headers and empty line to be `<HTML>` with *no* spaces before the opening left angle-bracket. If there are any other characters, they might not accept the output as HTML and print it as a plain text. Even if it works with your browser, it might not work for others.

One other approach is to use ‘here’ documents, e.g.:

```
print <<EOT;
<HTML>
<HEAD></HEAD>
<BODY>
EOT
```

Now let’s go back to the `$|=1` topic. I still disable buffering, for two reasons:

- I use relatively few `print()` calls. I achieve this by arranging for my `print()` statements to print multiline HTML, and not one line per `print()` statement.
- I want my users to see the output immediately. So if I am about to produce the results of a DB query which might take some time to complete, I want users to get some text while they are waiting. This improves the usability of my site. Ask yourself which 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 a while and then get the whole output at once, even if it’s a few milliseconds faster - assuming the browser didn’t time out during the wait.

An even better solution is to keep buffering enabled, and use a Perl API `rflush()` call to flush the buffers when needed. This way you can place the first part of the page that you are going to send to the user in the buffer, and flush it a moment before you are going to do some lengthy operation, like a DB query. So you kill two birds with one stone: you show some of the data to the user immediately, so she will feel that something is actually happening, and you have no performance hit from 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 each case.

Note: It might happen that some browsers do not render the page before they have received a significant amount. This is especially true if you insert `<link>` or `<script>` tags in your HTML header that require the browser to load a separate file. In that case, the user won't be able to see the content at once, no matter if you flush the buffers or not.

A workaround for this might be to use an output filter that replaces these tags with the files they refer to.

1.7.5 *Global vs. Fully Qualified Variables*

It's always a good idea to avoid using global variables where it's possible. Some variables must be either global, such as `@ISA` or else fully qualified such as `@MyModule::ISA`, so that Perl can see them from different packages.

A combination of `strict` and `vars` pragmas keeps modules clean and reduces a bit of noise. However, the `vars` pragma also creates aliases, as does `Exporter`, which eat up more memory. When possible, try to use fully qualified names instead of `use vars`.

For example write:

```
package MyPackage1;
use strict;
use vars; # added only for fair comparison
@MyPackage1::ISA = qw(CGI);
$MyPackage1::VERSION = "1.00";
1;
```

instead of:

```
package MyPackage2;
use strict;
use vars qw(@ISA $VERSION);
@ISA = qw(CGI);
$VERSION = "1.00";
1;
```

Note that we have added the `vars` pragma in the package that doesn't use it so the memory comparison will be fair.

Here are the numbers under Perl version 5.6.0

```
% perl -MGTop -MMyPackage1 -le 'print GTop->new->proc_mem($$)->size'
2023424
% perl -MGTop -MMyPackage2 -le 'print GTop->new->proc_mem($$)->size'
2031616
```

We have a difference of 8192 bytes. So every few global variables declared with `vars` pragma add about 8KB overhead.

Note that Perl 5.6.0 introduced a new `our()` pragma which works like `my()` scope-wise, but declares global variables.

```
package MyPackage3;
use strict;
use vars; # not needed, added only for fair comparison
our @ISA = qw(CGI);
our $VERSION = "1.00";
1;
```

which uses the same amount of memory as a fully qualified global variable:

```
% perl -MGTop -MMyPackage3 -le 'print GTop->new->proc_mem($$)->size'
2023424
```

Imported symbols act just like global variables, they can add up quick:

```
% perlbloat.pl 'use POSIX ()'
use POSIX () added 316k

% perlbloat.pl 'use POSIX'
use POSIX added 696k
```

That's 380k worth of aliases. Now let's say 6 different Apache::Registry scripts `'use POSIX;'` for `strftime()` or some other function: $6 * 380k = 2.3Mb$

One could save 2.3Mb per single process with `'use POSIX ();'` and using fully qualifying `POSIX::` function calls.

1.7.6 Object Methods Calls vs. Function Calls

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

1.7.6.1 The Overhead with Light Subroutines

Let's do some 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_call11.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 parameter; *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 as slow as the *function* call, 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. The function form has one parameter less to pass, less stack operations, less time to get to the guts of the subroutine.

perl5.6+ does better method caching, `Foo->method()` is a little bit faster (some constant folding magic), but not `Foo->$method()`. And the improvement does not address the @ISA lookup that still happens in either case.

1.7.6.2 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 significant is the time to perform the call, because the calling time is effectively fixed and is probably a very small overhead in comparison to the execution time of the method or function itself. Therefore the longer execution time of the function the smaller the relative overhead of the method call. The next benchmark proves this point:

```
bench_call12.pl
-----
package Foo;

use strict;
use Benchmark;

sub bar {
    my $class = shift;

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

1.7.6 Object Methods Calls vs. Function Calls

```
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 than *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 using 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, saving programmer time which, over the life of a project may turn out to be the most significant cost factor.

1.7.6.3 Are All Methods Slower than Functions?

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

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

vs

```
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 lays 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.

1.7.7 Imported Symbols and Memory Usage

There is a real memory hit when you import all of the functions 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.

1.7.7 Imported Symbols and Memory Usage

```
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 improvements, 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 core is written in C, giving it a significant memory and performance benefit. It has all the functionality of CGI.pm except the HTML generation functions.

1.7.8 Interpolation, Concatenation or List

Somewhat overlapping with the previous section we want to revisit the various approaches of mungling with strings, and compare the speed of using lists of strings compared to interpolation. We will add a string concatenation angle as well.

When the strings are small, it almost doesn't matter whether interpolation or a list is used. Here is a benchmark:

```
use Benchmark;
use Symbol;
my $fh = gensym;
open $fh, ">/dev/null" or die;

my($one, $two, $three, $four) = ('a'..'d');

timethese(1_000_000,
{
  interp => sub {
    print $fh "$one$two$three$four";
  },
  list => sub {
    print $fh $one, $two, $three, $four;
  },
  conc => sub {
    print $fh $one.$two.$three.$four;
  },
});

Benchmark: timing 1000000 iterations of conc, interp, list...
  conc:  3 wallclock secs ( 3.38 usr +  0.00 sys =  3.38 CPU)
  interp: 3 wallclock secs ( 3.45 usr + -0.01 sys =  3.44 CPU)
  list:  2 wallclock secs ( 2.58 usr +  0.00 sys =  2.58 CPU)
```

The concatenation technique is very similar to interpolation. The list technique is a little bit faster than interpolation. But when the strings are large, lists are significantly faster. We have seen this in the previous section and here is another benchmark to increase our confidence in our conclusion. This time we use 1000 character long strings:

```
use Benchmark;
use Symbol;
my $fh = gensym;
open $fh, ">/dev/null" or die;

my($one, $two, $three, $four) = map { $_ x 1000 } ('a'..'d');

timethese(500_000,
{
  interp => sub {
    print $fh "$one$two$three$four";
  },
  list => sub {
    print $fh $one, $two, $three, $four;
  },
});
```

1.7.9 Using Perl stat() Call's Cached Results

```
conc => sub {
    print $fh $one.$two.$three.$four;
},
});

Benchmark: timing 500000 iterations of interp, list...
conc:  5 wallclock secs ( 4.47 usr +  0.27 sys =  4.74 CPU)
interp: 4 wallclock secs ( 4.25 usr +  0.26 sys =  4.51 CPU)
list:  4 wallclock secs ( 2.87 usr +  0.16 sys =  3.03 CPU)
```

In this case using a list is about 30% faster than interpolation. Concatenation is a little bit slower than interpolation.

Let's look at this code:

```
$title = 'My Web Page';
print "<h1>$title</h1>";          # Interpolation (slow)
print '<h1>' . $title . '</h1>';  # Concatenation (slow)
print '<h1>', $title, '</h1>';    # List (fast for long strings)
```

When you use "`<h1>$title</h1>`" Perl does interpolation (since `" "` is an operator in Perl), which must parse the contents of the string and replace any variables or expressions it finds with their respective values. This uses more memory and is slower than using a list. Of course if there are no variables to interpolate it makes no difference whether to use `"string"` or `'string'`.

Concatenation is also potentially slow since Perl might create a temporary string which it then prints.

Lists are fast because Perl can simply deal with each element in turn. This is true if you don't run `join()` on the list at the end to create a single string from the elements of list. This operation might be slower than direct append to the string whenever a new string springs into existence.

[ReaderMETA]: Please send more `mod_perl` relevant Perl performance hints

1.7.9 Using Perl stat() Call's Cached Results

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

```
my $filename = "./test";
# three 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 before startup\n";
```

or the more efficient:

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

Two stat() calls were saved!

1.7.10 Optimizing Code

Here are some other resources that explain how to optimize your code, which are usually applied when you profile your code and need to optimize it but in many cases are useful to know when you develop the code.

- Interesting C code optimization notes, most applying to Perl code as well:
<http://www.uts.utoronto.ca/~harper/cscb09/lecture11.html#code>

[ReaderMETA]: please send me similar resources if you know of such.

1.8 Apache::Registry and Derivatives Specific Notes

These are the sections that deal solely with Apache::Registry and derived modules, like Apache::PerlRun and Apache::RegistryBB. No Perl handlers code is discussed here, so if you don't use these modules, feel free to skip this section.

1.8.1 Be Careful with Symbolic Links

As you know Apache::Registry caches the scripts in the packages whose names are constructed by scripts' URI. If you have the same script that can be reached by different URIs, which is possible if you have used symbolic links, you will get the same script stored twice in the memory.

For example:

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

Now the script can be reached through the both URIs */news/news.pl* and */news.pl*. It doesn't really matter until you advertise the two URIs, and users reach the same script from both of them.

So let's assume that you have issued the requests to the both URIs:

```
http://localhost/perl/news/news.pl
http://localhost/perl/news.pl
```

To spot the duplication you should use the Apache::Status module. Amongst other things, it shows all the compiled Apache::Registry scripts (using their respective packages):

If you are using the default configuration directives you should either use this URI:

```
http://localhost/perl-status?rgysubs
```

or just go to the main menu at:

```
http://localhost/perl-status
```

And click on Compiled Registry Scripts menu item.

META: we need a screen snapshot here!!!

If you the script was accessed through the URI that was remapped to the real file and through the URI that was remapped to the symbolic link, you will see the following output:

```
Apache::ROOT::perl::news::news_2ep1
Apache::ROOT::perl::news_2ep1
```

You should run the server in the single mode, to see it immediately. If you test it in the normal mode--it's possible that some child processes would show only one entry or none at all, since they might not serve the same requests as the others. For more hints see the section "Run the server in single mode".

1.9 Improving Performance by Prevention

There are two ways to improve performance: one is by tuning to squeeze the most out of your hardware and software; and the other is preventing certain bad things from happening, like impolite robots that crawl your site without pausing between requests, memory leakages, getting the memory unshared, making sure that some processes won't take up all the CPU etc.

In the following sections we are going to discuss about the tools and programming techniques that would help you to keep your service in order, even if you are not around.

1.9.1 *Memory leakage*

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

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

1.9.1.1 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 which served that script will grow by exactly that size. Now if you have 20 children, and all of them will serve this CGI, they will consume $20 \times 5\text{M} = 100\text{M}$ of RAM in total! If that's the case, try to use other approaches to processing the file, if possible. 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 you use a locking mechanism!

1.9.1.2 Copying Variables Between Functions

Now let's talk about 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 imaginary `process()` subroutine that processes the data and returns it. What happens if you pass the `$content` by value? You have just copied another 5M and the child has grown in size by **another** 5M. Watch your swap space! Now multiply it again by factor of 20 you have 200M of wasted RAM, which will apparently be reused, but it's a waste! Whenever you think the variable can grow bigger than a few Kb, pass it by reference!

Once I wrote a script that passed the contents of a little flat file database to a function that processed it by value -- it worked and it was fast, but after a time the database became bigger, so passing it by value was expensive. I had to make the 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 variables by reference, if a variable you are going to pass might eventually become bigger than you envisage at the time you code the program. 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 already
    # been modified
}
```

If you work with arrays or hashes it's:

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

We can still access individual elements of arrays and hashes that we have a reference to without dereferencing them:

```
$var_lr->[$index] get $index'th element of an array via a ref
$var_hr->{$key}    get $key'th element of a hash via a ref
```

For more information see `perldoc perlref`.

Another approach would be to use the `@_` array directly. This has the effect 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 subroutine, since it can confuse a potential user. It's not obvious that call like `process($content);` modifies the passed variable. Programmers (the users of your library in this case) are used to subroutines that either modify variables passed by reference or expressly return a result (e.g. `$content=process($content);`).

1.9.1.3 Work With Databases

If you do some DB processing, you will often encounter the need to read lots of records into your program, and then print them to the browser after they are formatted. I won'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 reference to the DBI module):

```

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

```

In the example above the `httpd_process` will grow 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 not to accumulate the records, but rather to 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 an HTML table with matched data, the only memory that is being used is a `@cols` array to hold temporary row values. The table will be rendered by the client browser only when the whole table will be out though.

```

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 similar attributes.

For those who think that `$sth->rows` will do the job, here is the quote from the DBI manpage:

```
rows();
```

```
$rv = $sth->rows;
```

Returns the number of rows affected by the last database altering command, or -1 if not known or not available. Generally you can only rely on a row count after a do or non-select execute (for some specific operations like update and delete) or after fetching all the rows of a select statement.

For select statements it is generally not possible to know how many rows will be returned except by fetching them all. Some drivers will return the number of rows the application has fetched so far but others may return -1 until all rows have been fetched. So use of the rows method with select statements is not recommended.

As a bonus, I wanted to write a single sub that flexibly processes any query. It would accept conditions, a call-back closure sub, select fields and restrictions.

```

# Usage:
# $o->dump(\%conditions,\&callback_closure,\@select_fields,@restrictions);
#
sub dump{
    my $self = shift;
    my %param = %{+shift}; # dereference hash
    my $rsub = shift;
    my @select_fields = @{+shift}; # dereference list
    my @restrict = shift || '';

    # create a list of cols values
    my @cols = ();
    @cols[0..$#select_fields] = ();

    my $do_sql = '';
    my @where = ();

    # make a @where list
    map { push @where, "$_='\$param{$_}\'" if $param{$_}; } keys %param;

```

1.9.1 Memory leakage

```
# prepare the sql statement
$do_sql = "SELECT ";
$do_sql .= join(" ", @restrict) if @restrict;      # append restriction list
$do_sql .= " " .join(",", @select_fields) ;      # append select list
$do_sql .= " FROM $DBConfig{TABLE} ";            # from table

# we will not add the WHERE clause if @where is empty
$do_sql .= " WHERE " . join " AND ", @where if @where;

print "SQL: $do_sql \n" if $debug;

$dbh->{RaiseError} = 1; # do this, or check every call for errors
$sth = $dbh->prepare($do_sql);
$sth->execute;
# Bind perl variables to columns.
$sth->bind_columns(undef,\(@cols));
while($sth->fetch) {
    &$rsub(@cols);
}
# print the tail or "no records found" message

# according to the previous calls
&$rsub();

} # end of sub dump
```

Now a callback closure sub can do lots of things. We need a closure to know what stage are we in: header, body or tail. For example, we want a callback closure for formatting the rows to print:

```
my $rsub = eval {
    # make a copy of @fields list, since it might go
    # out of scope when this closure is called
    my @fields = @fields;
    my @query_fields = qw(user dir tool act);    # no date field!!!
    my $header = 0;
    my $tail = 0;
    my $counter = 0;
    my %cols = ();                             # columns name=> value hash

    # Closure with the following behavior:
    # 1. Header's code will be executed on the first call only and
    #    if @_ was set
    # 2. Row's printing code will be executed on every call with @_ set
    # 3. Tail's code will be executed only if Header's code was
    #    printed and @_ isn't set
    # 4. "No record found" code will be executed if Header's code
    #    wasn't executed

    sub {
        # Header
        if (@_ and !$header){
            print "<TABLE>\n";
            print $q->Tr(map{ $q->td($_) } @fields );
            $header = 1;
        }

        # Body
```



```

if (@_) {
    print $q->Tr(map{$q->td($_)} @_ );
    $counter++;
    return;
}

# Tail, will be printed only at the end
if ($header and !($tail or @_)){
    print "</TABLE>\n $counter records found";
    $tail = 1;
    return;
}

# No record found
unless ($header){
    print $q->p($q->center($q->b("No record was found!\n")));
}

} # end of sub {}
}; # end of my $rsub = eval {

```

You might also want to check the section Preventing Your Processes from Growing and Limiting Other Resources Used by Apache Child Processes.

1.9.2 Preventing Your Processes from Growing

If you have already worked with `mod_perl`, you have probably noticed that it can be difficult to keep your `mod_perl` processes from using a lot of memory. The less memory you have, the fewer processes you can run and the worse your server will perform, especially under a heavy load. This chapter presents several common situations which can lead to unnecessary consumption of RAM, together with preventive measures.

When you need to control the size of your `httpd` processes, use one of the two modules `Apache::GTopLimit` and `Apache::SizeLimit` which kill Apache `httpd` processes when the latter grow too large or lose a big chunk of their shared memory. The two modules differ in methods for finding out the memory usage. `Apache::GTopLimit` relies on the *libgtop* library to perform this task, therefore if this library can be built on your platform you can use this module. `Apache::SizeLimit` includes different methods for different platforms, you will have to check the modules' manpage to figure out which platforms are supported.

1.9.2.1 Defining the Minimum Shared Memory Size Threshold

As we have already discussed, when it is first created an Apache child process usually has a large fraction of its memory shared with its parent. During the child process' life some of its data structures are modified and a part of its memory becomes unshared (pages become "*dirty*"), leading to an increase in memory consumption. You will remember that the `MaxRequestsPerChild` directive allows you to specify the number of requests a child process should serve before it is killed. One way to limit the memory consumption of a process is to kill it and let Apache replace it with a newly started process, which again will have all its memory shared with the Apache parent. The new child process serves requests and eventually the cycle is repeated.

This is a fairly crude means of limiting unshared memory and you will probably need to tune `MaxRequestsPerChild`, eventually finding an optimum value. If, as is likely, your service is undergoing constant changes then this is an inconvenient solution. You have to re-tune this number again and again to adapt to the ever changing code base.

You really want to set some guardian to watch the shared size and kill the process if it goes below some limit. This way, processes will not be killed unnecessarily.

To set a shared memory lower limit of 4MB using `Apache::GTopLimit` add the following code into the *startup.pl* file:

```
use Apache::GTopLimit;
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4096;
```

and in *httpd.conf*:

```
PerlFixupHandler Apache::GTopLimit
```

don't forget to restart the server for the changes to take effect.

This has the effect that as soon as the child process shares less than 4MB, (the corollary being that it must therefore be occupying a lot of memory with its unique pages), it will be killed after completing to serve the last request, and, as a consequence, a new child will take its place.

If you use `Apache::SizeLimit` you can accomplish the same with the adding to *startup.pl*:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MIN_SHARE_SIZE = 4096;
```

and in *httpd.conf*:

```
PerlFixupHandler Apache::SizeLimit
```

If you only want to set this limit for some requests (presumably the ones which you think are likely to cause memory to become unshared) then you can register a post-processing check using the `set_min_shared_size()` function. For example:

```
use Apache::GTopLimit;
if ($need_to_limit) {
    # make sure that at least 4MB are shared
    Apache::GTopLimit->set_min_shared_size(4096);
}
```

or for `Apache::SizeLimit`:

```
use Apache::SizeLimit;
if ($need_to_limit) {
    # make sure that at least 4MB are shared
    Apache::SizeLimit->setmin(4096);
}
```

Since accessing the process information adds a little overhead, you may want to only check the process size every N times. In this case set the `$Apache::GTopLimit::CHECK_EVERY_N_REQUESTS` variable. For example to test the size every other time, put in your *startup.pl*:

```
$Apache::GTopLimit::CHECK_EVERY_N_REQUESTS = 2;
```

or for `Apache::SizeLimit`:

```
$Apache::SizeLimit::CHECK_EVERY_N_REQUESTS = 2;
```

You can run the `Apache::GTopLimit` module in the debug mode by setting:

```
PerlSetVar Apache::GTopLimit::DEBUG 1
```

in *httpd.conf*. It's important that this setting should happen before the `Apache::GTopLimit` module is loaded.

When debug mode is turned *on* the module reports in the *error_log* file the memory usage of the current process and also when it detects that at least one of the thresholds was crosses and the process is going to be killed.

`Apache::SizeLimit` controls the debug level via `$Apache::SizeLimit::DEBUG` variable:

```
$Apache::SizeLimit::DEBUG = 1;
```

which can be modified any time, even after the module was loaded.

1.9.2.2 Potential Drawbacks of Memory Sharing Restriction

It's very important that the system won't be heavily engaged in swapping process. Some systems do swap in and out every so often even if they have plenty of real memory available and it's OK. The following applies to conditions when there is hardly any free memory available.

So if the system uses almost all of its real memory (including the cache), there is a danger of parent's process memory pages being swapped out (written to a swap device). If this happens the memory usage reporting tools will report all those swapped out pages as non-shared, even though in reality these pages are still shared on most OSs. When these pages are getting swapped in, the sharing will be reported back to normal after a certain amount of time. If a big chunk of the memory shared with child processes is swapped out, it's most likely that `Apache::SizeLimit` or `Apache::GTopLimit` will notice that the shared memory floor threshold was crossed and as a result kill those processes. If many of the parent process' pages are swapped out, and the newly created child process is already starting with shared memory below the limit, it'll be killed immediately after serving a single request (assuming that we the `$CHECK_EVERY_N_REQUESTS` is set to one). This is a very bad situation which will eventually lead to a state where the system won't respond at all, as it'll be heavily engaged in swapping process.

This effect may be less or more severe depending on the memory manager's implementation and it certainly varies from OS to OS, and different kernel versions. Therefore you should be aware of this potential problem and simply try to avoid situations where the system needs to swap at all, by adding more memory, reducing the number of child servers or spreading the load across more machines, if reducing the

number of child servers is not an options because of the request rate demands.

1.9.2.3 Defining the Maximum Memory Size Threshold

Not less important than maximizing shared memory is restricting the absolute size of the processes. If the processes grow after each request, and if nothing restricts them from growing, you can easily run out of memory.

Again you can set the `MaxRequestPerChild` directive to kill the processes after a few requests have been served. But as we have explained in the previous section this solution is not as good as one which monitors the process size and kills it only when some limit is reached.

If you have `Apache::GTopLimit` (described in the previous section) you can limit process' memory usage by setting the `$Apache::GTopLimit::MAX_PROCESS_SIZE` directive. For example if you want the processes to be killed when they reach 10MB you should put the following in your *startup.pl* file:

```
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10240;
```

Just as when limiting shared memory, you can set a limit for the current process using the `set_max_size()` method in your code:

```
use Apache::GTopLimit;
Apache::GTopLimit->set_max_size(10000);
```

For `Apache::SizeLimit` the equivalents are:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10240;
```

and:

```
use Apache::SizeLimit;
Apache::SizeLimit->setmax(10240);
```

1.9.2.4 Defining the Maximum Unshared Memory Size Threshold

Instead of setting the shared and total memory usage thresholds, you can set a single threshold which measures the amount of unshared memory, by subtracting the shared memory size from the total memory size.

Both modules allow you to set the thresholds in similar ways. With `Apache::GTopLimit` you can set the unshared memory threshold server-wide with:

```
$Apache::GTopLimit::MAX_PROCESS_UNSHARED_SIZE = 6144;
```

and locally for a handler with:

```
Apache::GTopLimit->set_max_unshared_size(6144);
```

If you are using `Apache::SizeLimit` the corresponding settings would be:

```
$Apache::SizeLimit::MAX_UNSHARED_SIZE = 6144;
```

and:

```
Apache::SizeLimit->setmax_unshared(6144);
```

1.9.3 Limiting Other Resources Used by Apache Child Processes

In addition to the absolute and shared memory sizes limiting, you might need to prevent the processes from excessive consumption of the system resources. Like limiting the CPU usage, the number of files that can be opened, or memory segment usage and more.

The `Apache::Resource` module allows this all by deploying the `BSD::Resource` module, which in turn uses the C function `setrlimit()` to set limits on system resources.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the CPU time or file size is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The `rlimit` structure is used to specify the hard and soft limits on a resource. (See the manpage for *setrlimit* for your OS specific information.)

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. So if you set `10:20`, the soft limit is 10 and the hard limit is 20. If you set just `10`--both the soft and the hard limits are set to 20.

The mostly spread usage of this module is to limit the CPU usage. The environment variable `PERL_RLIMIT_CPU` defines the maximum amount of CPU time the process can use. If it runs for longer than this, it gets killed, no matter what it does, either processing a new request or just waiting. This is very useful when you have a code with a bug and the process starts to spin in an infinite loop or alike using a lot of CPU and never completing the request.

META: verify this.

The value is measured in seconds. The following example sets the soft limit of the CPU usage to 120 seconds (the default is 360).

```
PerlModule Apache::Resource
PerlSetEnv PERL_RLIMIT_CPU 120
```

Of course you should tell `mod_perl` to use this module, which is done by adding the following directive to *httpd.conf*:

```
PerlChildInitHandler Apache::Resource
```

There are other resources that you might want to limit. For example you can limit the memory data and stack segment sizes (`PERL_RLIMIT_DATA` and `PERL_RLIMIT_STACK`), the maximum process file size (`PERL_RLIMIT_FSIZE`), the core file size (`PERL_RLIMIT_CORE`), the address space (virtual

memory) limit (`PERL_RLIMIT_AS`), etc. Refer to the `setrlimit(2)` man page on your OS for other possible resources. Remember to prepend `PERL_` before the resource types you will see in the man page.

If you configure `Apache::Status`, it will let you review the resources set in this way. Remember that `Apache::Status` must be loaded before `Apache::Resource` in order to enable the resources display menu.

If you want to set the debug mode set the `$Apache::Resource::Debug` before loading the module, for example by using the Perl sections in *httpd.conf*.

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

Now open in the *error_log* file using `tail` and watch the debug messages showing up, when the requests are served.

1.9.3.1 OS Specific notes

Note that under Linux `malloc()` uses `mmap()` instead of `brk()`. This is done to conserve virtual memory - that is, when you `malloc` a large block of memory, it isn't actually given to your program until you initialize it. The old-style `brk()` system call obeyed resource limits on data segment size as set in `setrlimit()` - `mmap()` doesn't.

`Apache::Resource`'s defaults put caps on data size and stack size. Linux's current memory allocation scheme doesn't honor these limits, so if you just do

```
PerlSetEnv PERL_RLIMIT_DEFAULTS On
PerlModule Apache::Resource
PerlChildInitHandler Apache::Resource
```

Your Apache processes are still free to use as much memory as they like.

However, `BSD::Resource` also has a limit called `RLIMIT_AS` (Address Space) which limits the total number of bytes of virtual memory assigned to a process. Happily, Linux's memory manager *does* honor this limit.

Therefore, you *can* limit memory usage under Linux with `Apache::Resource` -- simply add a line to *httpd.conf*:

```
PerlSetEnv PERL_RLIMIT_AS 67108864
```

This example sets a hard and soft limit of 64MB of total address space.

Refer to the `Apache::Resource` and `setrlimit(2)` manpages for more information.

1.9.4 Limiting the Number of Processes Serving the Same Resource

If you want to limit number of Apache children that could simultaneously be serving the (nearly) same resource, you should take a look at the `mod_throttle_access` module.

It solves the problem of too many concurrent request accessing the same URI, if for example the handler that serves this URI uses some resource that has a limitation on the maximum number of possible users or the handlers code is very CPU intensive and you cannot afford more than a certain number of concurrent requests to this specific URI.

Imagine that your service provides the three following URIs:

```
/perl/news/  
/perl/webmail/  
/perl/morphing/
```

The first two URIs are response critical as people want to read news and their email. The third URI is very CPU and RAM intensive image morphing service, provided as a bonus to your users. Since you don't want users to abuse this service, you have to set some limits on the number of concurrent requests for this resource, since if you don't--the other two critical resources can be hurt.

When you compile in and enable the Apache `mod_throttle_access` module, the `MaxConcurrentReqs` directive becomes available. For example, the following setting:

```
<Location "/perl/morphing">  
  <Limit PUT GET POST>  
    MaxConcurrentReqs 10  
  </Limit>  
</Location>
```

will allow only 10 concurrent PUT, GET or POST requests under the URI `/perl/morphing` to be processed at one time. The other two URIs remain unlimited.

1.9.5 Limiting the Request Rate Speed (Robot Blocking)

A limitation of using pattern matching to identify robots is that it only catches the robots that you know about, and then 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's `SpeedLimit` comes to your aid, see:

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

1.10 Perl Modules for Performance Improvement

These sections are about Perl modules that improve performance without requiring changes to your code. Mostly you just need to tweak the configuration file to plug these modules in.

1.10.1 Sending Plain HTML as Compressed Output

See `Apache::GzipChain` - compress HTML (or anything) in the `OutputChain`

1.10.2 Caching Components with `HTML::Mason`

META: complete the full description

`HTML::Mason` is a system that makes use of components to build HTML pages.

If most of your output is generated dynamically, but each finished page can be separated into different components, `HTML::Mason` can cache those components. This can really improve the performance of your service and reduce the load on the system.

Say for example that you have a page consisting of five components, each generated by a different SQL query, but for four of the five components it's the same four queries for each user so you don't have to rerun them again and again. Only one component is generated by a unique query and will not use the cache.

META: `HTML::Mason` docs (v 8.0) said `Mason` was 2-3 times slower than pure `mod_perl`, implying that the power & convenience made up for this.

META: Should also mention `Embperl` (especially since its C + XS)

1.11 Efficient Work with Databases under `mod_perl`

Most of the `mod_perl` enabled servers work with database engines, so in this section we will learn about two things: how `mod_perl` makes working with databases faster and a few tips for a more efficient DBI coding in Perl. (DBI provides an identical Perl interface to many database implementations.)

1.11.1 Persistent DB Connections

Another popular use of `mod_perl` is to take advantage of its ability to maintain persistent open database connections.

You want to have a persistent database connection because the most expensive part of a network transaction for most databases is the business of building and tearing down connections.

Of course the persistence doesn't help with the latency problems during the actual use of the database connections. Oracle is notoriously latency-sensitive which in most cases generates a network transaction per row returned which slows things down if the query execution matches many rows. You may want to

read the Tim Bunce's Advanced DBI talk at http://dbi.perl.org/doc/conferences/tim_1999/index.html which covers a lot of techniques to reduce latency.

So here is the basic approach of making the connection persistent:

```
# 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 connections 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 anything, the `Apache::DBI` module overloads the `disconnect()` method with a NOP. When a child exits 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
```

1.11.1.1 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 the DB for each child with the `connect_on_init()` method, thus saving 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 is 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 a `<Perl>` section. It will establish a connection when a child is started in that child process. See the `Apache::DBI` manpage for the requirements for this method.

1.11.1.2 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 (e.g PostgreSQL and Sybase) don't support caches of prepared plans. With Sybase you could open multiple connections to achieve the same result, although this is at the risk of getting deadlocks depending on what you are trying to do!

1.11.2 mod_perl Database Performance Improving

1.11.2.1 Analysis of the Problem

A common web application architecture is one or more application servers which handle requests from client browsers by consulting one or more database servers and performing a transform on the data. When an application must consult the database on every request, the interaction with the database server becomes the central performance issue. Spending a bit of time optimizing your database access can result in significant application performance improvements. In this analysis, a system using Apache, `mod_perl`, `DBI`, and Oracle will be considered. The application server uses Apache and `mod_perl` to service client requests, and `DBI` to communicate with a remote Oracle database.

In the course of servicing a typical client request, the application server must retrieve some data from the database and execute a stored procedure. There are several steps that need to be performed to complete the request:

- 1: Connect to the database server
- 2: Prepare a SQL SELECT statement
- 3: Execute the SELECT statement
- 4: Retrieve the results of the SELECT statement
- 5: Release the SELECT statement handle
- 6: Prepare a PL/SQL stored procedure call
- 7: Execute the stored procedure
- 8: Release the stored procedure statement handle
- 9: Commit or rollback
- 10: Disconnect from the database server

In this document, an application will be described which achieves maximum performance by eliminating some of the steps above and optimizing others.

1.11.2.2 Optimizing Database Connections

A naive implementation would perform steps 1 through 10 from above on every request. A portion of the source code might look like this:

```
# ...
my $dbh = DBI->connect('dbi:Oracle:host', 'user', 'pass')
|| die $DBI::errstr;

my $baz = $r->param('baz');

eval {
    my $sth = $dbh->prepare(qq{
        SELECT foo
        FROM bar
        WHERE baz = $baz
    });
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    $sth->finish;

    my $sph = $dbh->prepare(qq{
        BEGIN
            my_procedure(
                arg_in => $baz
            );
        END;
    });
    $sph->execute;
    $sph->finish;

    $dbh->commit;
};
if ($@) {
    $dbh->rollback;
```

```

}

$dbh->disconnect;
# ...

```

In practice, such an implementation would have hideous performance problems. The majority of the execution time of this program would likely be spent connecting to the database. An examination shows that step 1 is comprised of many smaller steps:

```

1: Connect to the database server
1a: Build client-side data structures for an Oracle connection
1b: Look up the server's alias in a file
1c: Look up the server's hostname
1d: Build a socket to the server
1e: Build server-side data structures for this connection

```

The naive implementation waits for all of these steps to happen, and then throws away the database connection when it is done! This is obviously wasteful, and easily rectified. The best solution is to hoist the database connection step out of the per-request lifecycle so that more than one request can use the same database connection. This can be done by connecting to the database server once, and then not disconnecting until the Apache child process exits. The `Apache::DBI` module does this transparently and automatically with little effort on the part of the programmer.

`Apache::DBI` intercepts calls to DBI's connect and disconnect methods and replaces them with its own. `Apache::DBI` caches database connections when they are first opened, and it ignores disconnect commands. When an application tries to connect to the same database, `Apache::DBI` returns a cached connection, thus saving the significant time penalty of repeatedly connecting to the database. You will find a full treatment of `Apache::DBI` at [Persistent DB Connections](#)

When `Apache::DBI` is in use, none of the code in the example needs to change. The code is upgraded from naive to respectable with the use of a simple module! The first and biggest database performance problem is quickly dispensed with.

1.11.2.3 Utilizing the Database Server's Cache

Most database servers, including Oracle, utilize a cache to improve the performance of recently seen queries. The cache is keyed on the SQL statement. If a statement is identical to a previously seen statement, the execution plan for the previous statement is reused. This can be a considerable improvement over building a new statement execution plan.

Our respectable implementation from the last section is not making use of this caching ability. It is preparing the statement:

```
SELECT foo FROM bar WHERE baz = $baz
```

The problem is that `$baz` is being read from an HTML form, and is therefore likely to change on every request. When the database server sees this statement, it is going to look like:

```
SELECT foo FROM bar WHERE baz = 1
```

and on the next request, the SQL will be:

```
SELECT foo FROM bar WHERE baz = 42
```

Since the statements are different, the database server will not be able to reuse its execution plan, and will proceed to make another one. This defeats the purpose of the SQL statement cache.

The application server needs to make sure that SQL statements which are the same look the same. The way to achieve this is to use placeholders and bound parameters. The placeholder is a blank in the SQL statement, which tells the database server that the value will be filled in later. The bound parameter is the value which is inserted into the blank before the statement is executed.

With placeholders, the SQL statement looks like:

```
SELECT foo FROM bar WHERE baz = :baz
```

Regardless of whether baz is 1 or 42, the SQL always looks the same, and the database server can reuse its cached execution plan for this statement. This technique has eliminated the execution plan generation penalty from the per-request runtime. The potential performance improvement from this optimization could range from modest to very significant.

Here is the updated code fragment which employs this optimization:

```
# ...
my $dbh = DBI->connect('dbi:Oracle:host', 'user', 'pass')
|| die $DBI::errstr;

my $baz = $r->param('baz');

eval {
    my $sth = $dbh->prepare(qq{
        SELECT foo
          FROM bar
         WHERE baz = :baz
    });
    $sth->bind_param(':baz', $baz);
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    $sth->finish;

    my $sph = $dbh->prepare(qq{
        BEGIN
            my_procedure(
                arg_in => :baz
            );
        END;
    });
    $sph->bind_param(':baz', $baz);
```

```

    $sph->execute;
    $sph->finish;

    $dbh->commit;
};
if ($@) {
    $dbh->rollback;
}
# ...

```

1.11.2.4 Eliminating SQL Statement Parsing

The example program has certainly come a long way and the performance is now probably much better than that of the first revision. However, there is still more speed that can be wrung out of this server architecture. The last bottleneck is in SQL statement parsing. Every time DBI's `prepare()` method is called, DBI parses the SQL command looking for placeholder strings, and does some housekeeping work. Worse, a context has to be built on the client and server sides of the connection which the database will use to refer to the statement. These things take time, and by eliminating these steps the time can be saved.

To get rid of the statement handle construction and statement parsing penalties, we could use DBI's `prepare_cached()` method. This method compares the SQL statement to others that have already been executed. If there is a match, the cached statement handle is returned. But the application server is still spending time calling an object method (very expensive in Perl), and doing a hash lookup. Both of these steps are unnecessary, since the SQL is very likely to be static and known at compile time. The smart programmer can take advantage of these two attributes to gain better database performance. In this example, the database statements will be prepared immediately after the connection to the database is made, and they will be cached in package scalars to eliminate the method call.

What is needed is a routine that will connect to the database and prepare the statements. Since the statements are dependent upon the connection, the integrity of the connection needs to be checked before using the statements, and a reconnection should be attempted if needed. Since the routine presented here does everything that `Apache::DBI` does, it does not use `Apache::DBI` and therefore has the added benefit of eliminating a cache lookup on the connection.

Here is an example of such a package:

```

package My::DB;

use strict;
use DBI ();

sub connect {
    if (defined $My::DB::conn) {
        eval {
            $My::DB::conn->ping;
        };
        if (!$@) {
            return $My::DB::conn;
        }
    }

    $My::DB::conn = DBI->connect(

```

```

        'dbi:Oracle:server', 'user', 'pass', {
            PrintError => 1,
            RaiseError => 1,
            AutoCommit => 0
        }
    ) || die $DBI::errstr; #Assume application handles this

    $My::DB::select = $My::DB::conn->prepare(q{
        SELECT foo
        FROM bar
        WHERE baz = :baz
    });

    $My::DB::procedure = $My::DB::conn->prepare(q{
        BEGIN
            my_procedure(
                arg_in => :baz
            );
        END;
    });

    return $My::DB::conn;
}

1;

```

Now the example program needs to be modified to use this package.

```

# ...
my $dbh = My::DB->connect;

my $baz = $r->param('baz');

eval {
    my $sth = $My::DB::select;
    $sth->bind_param(':baz', $baz);
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    my $sph = $My::DB::procedure;
    $sph->bind_param(':baz', $baz);
    $sph->execute;

    $dbh->commit;
};
if ($@) {
    $dbh->rollback;
}
# ...

```

Notice that several improvements have been made. Since the statement handles have a longer life than the request, there is no need for each request to prepare the statement, and no need to call the statement handle's finish method. Since `Apache::DBI` and the `prepare_cached()` method are not used, no cache

lookups are needed.

1.11.2.5 Conclusion

The number of steps needed to service the request in the example system has been reduced significantly. In addition, the hidden cost of building and tearing down statement handles and of creating query execution plans is removed. Compare the new sequence with the original:

```
1: Check connection to database
2: Bind parameter to SQL SELECT statement
3: Execute SELECT statement
4: Fetch rows
5: Bind parameters to PL/SQL stored procedure
6: Execute PL/SQL stored procedure
7: Commit or rollback
```

It is probably possible to optimize this example even further, but I have not tried. It is very likely that the time could be better spent improving your database indexing scheme or web server buffering and load balancing.

1.12 Using 3rd Party Applications

It's been said that no one can do everything well, but one can do something specific extremely well. This seems to be true for many software applications, when you don't try to do everything but instead concentrate on something specific you can do it really well.

Based on the above introduction, while the `mod_perl` server can do many many things, there are other applications (or Apache server modules) that can do some specific operations faster or do a really great job for the `mod_perl` server by unloading it when doing some operations by themselves.

Let's take a look at a few of these.

1.12.1 Proxying the *mod_perl* Server

Proxy gives you a great performance increase in most cases. It's discussed in the section Adding a Proxy Server in `http Accelerator Mode`.

1.13 Upload and Download of Big Files

You don't want to tie up your precious `mod_perl` backend server children doing something as long and simple as transferring a file, especially a big one. The overhead saved by `mod_perl` is typically under one second, which is an enormous saving for the scripts whose run time is under one second. The user won't really see any important performance benefits from `mod_perl`, since the upload may take up to several minutes.

If some particular script's main functionality is the uploading or downloading of big files, you probably want it to be executed on a plain apache server under mod_cgi (i.e. performing this operation on the front-end server, if you use a dual-server setup).

This of course assumes that the script requires none of the functionality of the mod_perl server, such as custom authentication handlers.

1.14 Apache/mod_perl Build Options

It's important how you build mod_perl enabled Apache. It influences the size of the httpd executable, some irrelevant modules might slow the performance.

[ReaderMETA: Any other building time things that influence performance?]

1.14.1 mod_perl Process Size as a Function of Compiled in C Modules and mod_perl Features

You might wonder whether it's better to compile in only the required modules and mod_perl hooks, or it doesn't really matter. To answer on this question lets first make a few compilation and compare the results.

So we are going to build mod_perl starting with:

```
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
    DO_HTTPD=1 USE_APACI=1
```

and followed by one of these option groups:

1. Default

no arguments

2. Minimum

```
APACI_ARGS='--disable-module=env, \
    --disable-module=negotiation, \
    --disable-module=status, \
    --disable-module=info, \
    --disable-module=include, \
    --disable-module=autoindex, \
    --disable-module=dir, \
    --disable-module=cgi, \
    --disable-module=asis, \
    --disable-module=imap, \
    --disable-module=userdir, \
    --disable-module=access, \
    --disable-module=auth'
```

3. Everything

```
EVERYTHING=1
```

4. Everything + Debug

```
EVERYTHING=1 PERL_DEBUG=1
```

After re-compiling with arguments of each of these groups, we can summarize the results:

Build group	httpd size (bytes)	Difference
Minimum	892928	+ 0
Default	994316	+101388
Everything	1044432	+151504
Everything+Debug	1162100	+269172

Indeed when you strip most of the default things, the server size is slimmer. But the savings are insignificant since you don't multiply the added size by the number of child processes if your OS supports sharing memory. The parent processes is a little bigger, but it shares these memory pages with its child processes. Of course not everything will be shared, if some module you add does some process memory modification particular to the process, but the most will.

And of course this was just an example to show the difference in size. It doesn't mean that you can everything away, since there will be Apache modules and mod_perl options that you won't be able to work without.

But as a good system administrator's rule says: *"Run the absolute minimum of the applications. If you don't know or need something, disable it"*. Following this rule to decide on the required Apache components and disabling the unneeded default components, makes you a good Apache administrator.

1.15 Perl Build Options

The Perl interpreter lays in the brain of the mod_perl server and if we can optimize perl into doing things faster under mod_perl we make the whole server faster. Generally, optimizing the Perl interpreter means enabling or disabling some command line options. Let's see a few important ones.

1.15.1 *-DTWO_POT_OPTIMIZE and -DPACK_MALLOC Perl Build Options*

Newer Perl versions also have build time options to reduce runtime memory consumption. These options might shrink the size of your httpd by about 150k -- quite a big number if you remember to multiply it by the number of children you use.

The `-DTWO_POT_OPTIMIZE` macro improves allocations of data with size close to a power of two; but this works for big allocations (starting with 16K by default). Such allocations are typical for big hashes and special-purpose scripts, especially image processing.

Perl memory allocation is by bucket with sizes close to powers of two. Because of these the malloc() overhead may be big, especially for data of size exactly a power of two. If `PACK_MALLOC` is defined, perl uses a slightly different algorithm for small allocations (up to 64 bytes long), which makes it possible to have overhead down to 1 byte for allocations which are powers of two (and appear quite often).

Expected memory savings (with 8-byte alignment in `alignbytes`) is about 20% for typical Perl usage. Expected slowdown due to additional malloc() overhead is in fractions of a percent and hard to measure, because of the effect of saved memory on speed.

You will find these and other memory improvement details in `perl5004delta.pod`.

Important: both options are On by default in perl versions 5.005 and higher.

1.15.2 *-Dusemymalloc Perl Build Option*

You have a choice to use the native or Perl's own malloc() implementation. The choice depends on your Operating System. Unless you know which of the two is better on yours, you better try both and compare the benchmarks.

To build without Perl's malloc(), you can use the Configure command:

```
% sh Configure -Uusemymalloc"
```

Note that:

```
-U == undefine usemymalloc (use system malloc)
-D == define    usemymalloc (use Perl's malloc)
```

It seems that Linux still defaults to system malloc so you might want to configure Perl with `-Dusemymalloc`. Perl's malloc is not much of a win under linux, but makes a **huge** difference under Solaris.

1.16 Architecture Specific Compile Options

When you build Apache and Perl you can optimize the compiled applications to take the benefits of your machine's architecture.

Everything depends on the kind of compiler that you use, the kind of CPU and

For example if you use gcc(1) you might want to use:

- `-march=pentium` if you have a pentium CPU
- `-march=pentiumpro` for pentiumpro and above (but the binary won't run on i386)
- `-fomit-frame-pointer` makes extra register available but disables debugging

- you can try these options were reported to improve the performance: *-ffast-math*, *-malign-double*, *-funroll-all-loops*, *-fno-rtti*, *-fno-exceptions*.

see the gcc(1) manpage for the details about these

- and of course you may want to change the usually default *-O2* flag with a higher number like *-O3*. *-OX* (where X is a number between 1 and 6) defines a collection of various optimization flags, the higher the number the more flags are bundled. The gcc man page will tell you what flags are used for each number.

Test your applications thoroughly when you change the default optimization flags, especially when you go beyond *-O2*. It's possible that the optimization will make the code work incorrectly and/or cause segmentation faults.

See your preferred compiler's man page for detailed information about optimization.

1.17 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.18 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

1	Performance Tuning	1
1.1	Description	2
1.2	The Big Picture	2
1.3	System Analysis	3
1.3.1	Software Requirements	3
1.3.2	Hardware Requirements	3
1.4	Essential Tools	4
1.4.1	Benchmarking Applications	4
1.4.1.1	Benchmarking Perl Code	4
1.4.1.2	Benchmarking a Graphic Hits Counter with Persistent DB Connections	5
1.4.1.3	Benchmarking Response Times	5
1.4.1.3.1	ApacheBench	5
1.4.1.3.2	httperf	6
1.4.1.3.3	http_load	7
1.4.1.3.4	the crashme Script	8
1.4.1.4	Benchmarking PerlHandlers	11
1.4.1.5	Other Benchmarking Tools	11
1.4.2	Code Profiling Techniques	12
1.4.3	Measuring the Memory of the Process	16
1.4.4	Measuring the Memory Usage of Subroutines	18
1.5	Know Your Operating System	21
1.5.1	Sharing Memory	21
1.5.1.1	How Shared Is My Memory?	22
1.5.1.2	Calculating Real Memory Usage	22
1.5.1.3	Are My Variables Shared?	23
1.5.1.4	Preloading Perl Modules at Server Startup	27
1.5.1.5	Preloading Registry Scripts at Server Startup	30
1.5.1.6	Modules Initializing at Server Startup	31
1.5.1.6.1	Initializing DBI.pm	31
1.5.1.6.2	Initializing CGI.pm	35
1.5.2	Increasing Shared Memory With mergemem	37
1.5.3	Forking and Executing Subprocesses from mod_perl	37
1.5.3.1	Forking a New Process	38
1.5.3.2	Freeing the Parent Process	39
1.5.3.3	Detaching the Forked Process	40
1.5.3.4	Avoiding Zombie Processes	40
1.5.3.5	A Complete Fork Example	42
1.5.3.6	Starting a Long Running External Program	44
1.5.3.7	Starting a Short Running External Program	46
1.5.3.8	Executing system() or exec() in the Right Way	47
1.5.4	OS Specific Parameters for Proxying	47
1.6	Performance Tuning by Tweaking Apache Configuration	47
1.6.1	Configuration Tuning with ApacheBench	48
1.6.2	Choosing MaxClients	53

Table of Contents:

1.6.3	Choosing MaxRequestsPerChild	55
1.6.4	Choosing MinSpareServers, MaxSpareServers and StartServers	56
1.6.5	Summary of Benchmarking to tune all 5 parameters	56
1.6.6	KeepAlive	58
1.6.7	PerlSetupEnv Off	59
1.6.8	Reducing the Number of stat() Calls Made by Apache	60
1.7	TMTOWTDI: Convenience and Habit vs. Performance	64
1.7.1	Apache::Registry PerlHandler vs. Custom PerlHandler	65
1.7.2	"Bloatware" modules	68
1.7.3	Apache::args vs. Apache::Request::param vs. CGI::param	70
1.7.4	Using \$ =1 Under mod_perl and Better print() Techniques.	73
1.7.5	Global vs. Fully Qualified Variables	75
1.7.6	Object Methods Calls vs. Function Calls	76
1.7.6.1	The Overhead with Light Subroutines	76
1.7.6.2	The Overhead with Heavy Subroutines	77
1.7.6.3	Are All Methods Slower than Functions?	78
1.7.7	Imported Symbols and Memory Usage	79
1.7.8	Interpolation, Concatenation or List	81
1.7.9	Using Perl stat() Call's Cached Results	82
1.7.10	Optimizing Code	83
1.8	Apache::Registry and Derivatives Specific Notes	83
1.8.1	Be Careful with Symbolic Links	83
1.9	Improving Performance by Prevention	84
1.9.1	Memory leakage	84
1.9.1.1	Reading In A Whole File	84
1.9.1.2	Copying Variables Between Functions	85
1.9.1.3	Work With Databases	86
1.9.2	Preventing Your Processes from Growing	89
1.9.2.1	Defining the Minimum Shared Memory Size Threshold	89
1.9.2.2	Potential Drawbacks of Memory Sharing Restriction	91
1.9.2.3	Defining the Maximum Memory Size Threshold	92
1.9.2.4	Defining the Maximum Unshared Memory Size Threshold	92
1.9.3	Limiting Other Resources Used by Apache Child Processes	93
1.9.3.1	OS Specific notes	94
1.9.4	Limiting the Number of Processes Serving the Same Resource	95
1.9.5	Limiting the Request Rate Speed (Robot Blocking)	95
1.10	Perl Modules for Performance Improvement	96
1.10.1	Sending Plain HTML as Compressed Output	96
1.10.2	Caching Components with HTML::Mason	96
1.11	Efficient Work with Databases under mod_perl	96
1.11.1	Persistent DB Connections	96
1.11.1.1	Preopening Connections at the Child Process' Fork Time	97
1.11.1.2	Caching prepare() Statements	98
1.11.2	mod_perl Database Performance Improving	98
1.11.2.1	Analysis of the Problem	98
1.11.2.2	Optimizing Database Connections	99
1.11.2.3	Utilizing the Database Server's Cache	100

1.11.2.4 Eliminating SQL Statement Parsing	102
1.11.2.5 Conclusion	104
1.12 Using 3rd Party Applications	104
1.12.1 Proxying the mod_perl Server	104
1.13 Upload and Download of Big Files	104
1.14 Apache/mod_perl Build Options	105
1.14.1 mod_perl Process Size as a Function of Compiled in C Modules and mod_perl Features	105
1.15 Perl Build Options	106
1.15.1 -DTWO_POT_OPTIMIZE and -DPACK_MALLOC Perl Build Options	106
1.15.2 -Dusemymalloc Perl Build Option	107
1.16 Architecture Specific Compile Options	107
1.17 Maintainers	108
1.18 Authors	108