Accelerate Your Server

Delivering Web Content Faster with mod_perl 2

Apache 2, mod_perl 2

- Discussing Apache version 2
- Apache 2 is the latest version of the Apache web server
- mod_perl 2 is the latest version of mod_perl, built for Apache 2
- General concepts apply to Apache 1 and mod_perl 1

CGI

- You have some existing CGI programs
- It becomes popular
- Now what?

Install mod_perl!

- Use your existing CGI scripts
- Get order of magnitude better performance
- Continue to develop in perl

CGI

- What's slow?

CGI

- httpd fork()s
- httpd exec(../cgi-bin/myscript.pl)
- OS sees #!/usr/bin/perl in .pl file
- OS fork()s
- OS exec()s /usr/bin/perl
- Perl opens, inits, parses myscript.pl
- Perl runs the script

CGI

- That's a bunch of work...
- ...for each request

mod_perl

- Apache has an embedded Perl interpreter
- Starts on server start-up
- Can load your code on startup
- No forks if a child process is available
- Finds the right sub and runs it

Getting Started

Install mod_perl

- mod_perl 2 for Apache 2
- (mod_perl 1 for Apache 1)

Install mod_perl

- Source:
- http://perl.apache.org/dist/mod_perl-2.0-current.tar.gz
- Instructions:
- http://perl.apache.org/docs/2.0/user/install/install.html

Install mod_perl

- unpack and cd to mod_perl dir
  - perl Makefile.PL MP_APXS=/path/to/apxs
  - make
  - make test
  - make install

(instructions for Windows on web)

Configure Apache

```
LoadModule cgi_module modules/mod_cgi.so
LoadModule perl_module modules/mod_perl.so

Alias /cgi-bin /var/www/cgi-bin
Alias /perlrun /var/www/cgi-bin
```

```
<Location /cgi-bin>
 SetHandler cgi-script
 Option +ExecCGI
</Location>
<Location /perlrun>
 SetHandler perl-script
 Option +ExecCGI
 PerlHandler ModPerl::PerlRun
</Location>
```

We're Running

```
#!/usr/bin/perl

use CGI qw(:standard);

print header();

for my $key ( sort keys %ENV ){
 if( $key =~ /perl/ or $ENV{$key} =~ /perl/ ){
  print '<p>' . $key . '=>' . $ENV{$key} . '</p>';
 }
}
```

We're Running

MOD_PERL=>mod_perl/2.0.3

SERVER_SOFTWARE=>Apache/2.2.8 (Unix) mod_apreq2-20051231/2.6.0 mod_perl/2.0.3 Perl/v5.8.8

ModPerl::PerlRun

ModPerl::Registry

ModPerl::PerlRun

- Closest to CGI emulation
- Persistent perl, reloaded code
- On every request for the script:
  - Found, loaded, parsed, BEGIN {}, Executed, END {}, Destroyed

ModPerl::Registry

- Less true to CGI emulation
- Persistent perl, persistent code
- Faster than PerlRun
- On every request:
  - Use code in cache (unless not found or disk is newer), `BEGIN {}`, executed, `END {}`

ModPerl::Registry

- Try Registry first
- If you have problems, try PerlRun

How Fast?

- Run ab, apache benchmark, on each script...

>ab -c1 -n100 http://localhost:8081/[method]/speed.cgi

CGI:
Requests per second: 2.03 [#/sec] (mean)
Time per request: 491.962 [ms] (mean)
Transfer rate: 0.51 [Kbytes/sec] received

PerlRun:
Requests per second: 27.61 [#/sec] (mean)
Time per request: 36.222 [ms] (mean)
Transfer rate: 7.18 [Kbytes/sec] received

Registry:
Requests per second: 228.91 [#/sec] (mean)
Time per request: 4.369 [ms] (mean)
Transfer rate: 59.52 [Kbytes/sec] received

CGI Problem Areas

Global Variables

- CGI scripts run OK with global variables
- mod_perl has persistent perl
- global is very global and persistent
- Don’t use globals
- Scope everything with ‘my’

Global Variables

- Be sure to ‘my’ variables in subs too
- Watch out for “Variable "$foo" will not stay shared”

cwd Behavior

- Current working directory (cwd) not the same under mod_perl
- Relative paths won’t work:

  require “mylibs.pl”;

  “Can't locate mylibs.pl in @INC...”

cwd Behavior

- Use full paths (best and safe with threads)
- If your MPM is Prefork:
  - ModPerl::RegistryPrefork
  - ModPerl::PerlRunPrefork

General Tips

- always ‘use strict’
- always ‘use warnings’
- watch your error_log
- scope everything with ‘my’

Memory

Trade-offs

- Q: What did we trade to get such speed?
- A: Memory

```
CGI:
 PID  RSS
54126 3392
54127 2012

PerlRun:
 PID  RSS
54160 3392
54161 22192

Registry:
 PID  RSS
54223 3392
54224 22180
```

Memory Loss

- No, not forgetting things
- If you run out of memory, you'll start paging
- Performance gains out the window

Dual Servers

- Speed allows small number of mod_perl servers to serve many requests
- Memory is wasted if processes are doing other things
- Other things include serving static content and feeding content to client

Dual Servers

- Run 2 Apache servers
- Back (application) server is the mod_perl server generating dynamic content
- Front (proxy) server delivers static content and buffers content back to clients

Dual Servers

- Typical configuration might be 50 (or more) front servers and 5 back servers
- 50 x 3MB = 150MB
- 5 x 25MB = 125MB

Dual Servers

- Other tools for front end (e.g. Squid)
- Each front server option has pros/cons
- Plan for a 2-server architecture: pick the front server based on needs
- If not sure, Apache is easy because it's already installed

Database

What else is slow?

- Looking for more slow operations
- Most dynamic sites use a backend DB
- Connecting to a database is slow

DB Connection

- Typical CGI calls connect
- Perform a bunch of DB operations
- Disconnect
- Next request, connect to same DB again with same username/password

DB Connection

- Use Apache::DBI to persist the DB connection

  # Configuration in httpd.conf or startup.pl. This comes before all other modules using DBI

  PerlModule Apache::DBI

- Everything else is transparent

DB Connection

- Watches for your connect
- Hashes the connect with connect params and all attributes (RaiseError, etc.)
- Intercepts the disconnect
- On next connect, returns the cached DB handle

DB Connection

- One connection per Apache child, per set of connect attributes
- Need to watch the number of connections to the DB
- Another reason to limit the number of mod_perl children
- Newer modules offer connection pools (DBD::Gofer)

DB Connection

- Apache child has persistent session, so commit/rollback if you use transactions
- Apache::DBI pings the connection and reconnects if the session has been disconnected

Summary

- mod_perl gives your CGIs amazing speed
- cost is memory
- stricter coding practices
- persistent DB connections

Next Steps

- mod_perl handles the slow bits in your environment
- Remaining slowness is your code (or network, etc.)
- Benchmark, isolate slowness, and try to optimize
- After that, it takes as long as it takes

Questions on server acceleration

Thanks

- “From CGI to mod_perl 2.0, Fast!” by Philippe M. Chiasson http://gozer.ectoplasm.org/talks/

Extending Apache with mod_perl

Extending Apache

- Apache web server is designed with an open API
- Added functionality provided via modules: mod_ssl, mod_cgi, etc.
- The API was designed for C programmers
- Many modules are written in C

Extending Apache

- Typical workflow for C module:
  - prepare C source files
  - compile into object files
  - link or move to DSO directory
  - modify Apache configuration
  - start server and test

“This sounds like a drag, and it is.”

-- “Writing Apache Modules with Perl and C,” Lincoln Stein & Doug MacEachern, O’Reilly 1999.

Extending Apache

- mod_perl allows you to write Apache modules for any part of Apache using Perl
- Full access to the Apache API
- Full access to the request data from and to the client

Extending Apache

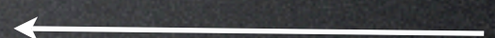
- Much quicker development cycle
- Apache::Test for testing
- CPAN modules

- Easy things can be easy and hard things possible
- Can still re-write in C for extra ounce of performance

Apache Request Cycle

- Apache processes requests using many different phases
- There are hooks into each of these phases to allow you to control behavior
- Generating content (creating a dynamic web page) is only one phase

HTTP Request Cycle

1. PerlPostReadRequestHandler (PerlInitHandler)
2. PerlTransHandler
3. PerlMapToStorageHandler
4. PerlHeaderParserHandler (PerlInitHandler)
5. PerlAccessHandler
6. PerlAuthenHandler
7. PerlAuthzHandler
8. PerlTypeHandler
9. PerlFixupHandler
10. PerlResponseHandler 
11. PerlLogHandler
12. PerlCleanupHandler

Response Handler

- In mod_perl, you use a response handler to generate content directly in Apache
- Apache doesn't fork an external process, like with CGI
- Where most people work when they do web development

mod_perl 2 Handler Basics

Handler Basics

- Q: What is a handler?
- A: A typical handler is a Perl package that contains a ‘handler’ subroutine.

```
package MyApache2::CurrentTime;
use strict;
use warnings;
use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::Const -compile => qw(OK);

sub handler {
  my $r = shift;
  $r->content_type('text/plain');
  $r->print("Now is: ", scalar(localtime) , "\n");
  return Apache2::Const::OK;
}
1;
```

Handler Basics

- Put the module where Apache can find it, by default Perl's @INC and HTTPD_ROOT.
- Tell Apache what requests should be handled by your module

```
# Define when it should run
<Location /time>
    SetHandler modperl
    PerlResponseHandler MyApache2::
CurrentTime
</Location>
```

Handler Arguments

- First argument depends on handler type (for function handlers)
- HTTP response phase argument is an Apache2::RequestRec object
- Usually called $r
- Useful hook to many Apache values

Method Handlers

- Can write handlers using OO Perl
- Use the ‘method’ attribute with the subroutine
- mod_perl passes the class as the first argument

```
package Bird::Eagle;
@ISA = qw(Bird); # Or use base

sub handler : method {
  my ($class_or_object, $r) = @_;
  # Some code
}

sub new { bless {}, __PACKAGE__ }
```

Return Values

- Apache expects certain return values, provided by Apache2::Const
- Apache2::Const::OK tells Apache the handler has successfully finished
- Apache2::Const::DECLINED indicates handler was not interested in request
- Valid return codes depend on phase

Other Phases

Extending Apache

- What parts are there besides content delivery?
- Which of these parts of the cycle can you modify or extend with mod_perl?

HTTP Request Cycle

1. PerlPostReadRequestHandler (PerlInitHandler)
2. PerlTransHandler
3. PerlMapToStorageHandler
4. PerlHeaderParserHandler (PerlInitHandler)
5. PerlAccessHandler
6. PerlAuthenHandler
7. PerlAuthzHandler
8. PerlTypeHandler
9. PerlFixupHandler
10. PerlResponseHandler
11. PerlLogHandler
12. PerlCleanupHandler

Other Phases

- Apache evaluates requests through many phases before you generate content
- AAA phases, for example, are where Apache evaluates access, authentication, and authorization

Other Phases

- Can modify server behavior at the right location
- Can write custom modules to handle special conditions
- Can seamlessly interact with Apache in these cycles
- All in perl

Example:
Auth Change

Auth Change

- Running mod_auth_dce
- Switched to mod_auth_ldap
  - Some subtle behaviors were different:
    - Allowed blank spaces
    - Case insensitive

Auth Change

- After the change, users could pass auth, but not be found in the user database
- Could fix the username when looking up in the DB
- But then the users think an incorrect username works

Auth Change

- Better solution: write a handler to reject these bad usernames the same way mod_auth_dce did
- More accurate response for users
- Less resources on the server
- Handled at the proper phase

```
PerlAuthenHandler MyAuth # In httpd.conf

## Module
 sub handler{
   my $r = shift;
   $r->get_basic_auth_pw; # Verify auth

   if (looks_like_a_username($r->user)) {
      return Apache2::Const::DECLINED; # Allow
others to run
   }
   else {
     # not a valid username
    $r->note_basic_auth_failure;
    return Apache2::Const::AUTH_REQUIRED;
   }
  }
```

Execution Order

- We wanted our handler to run first
- Then run mod_auth_ldap
- Can still run both, in the order you want

Execution Order

- Behavior depends on the phase
- Three types: VOID, RUN_FIRST, RUN_ALL
- Multiple handlers registered for a phase are called stacked handlers
- See documentation for run type and expected return values

Auth Change

- With very little code, we were able to modify the auth behavior
- Modification ran at the correct phase
- Integrated with Apache so the rest of the cycle operated normally

Example:
Site Redesign

Website Redesign

- Site goes from static to dynamic
- Many published links to articles like: http://example.com/news/20021031/09/index.html
- Now URIs look like: http://example.com/perl/news.pl?date=20021031;id=09;page=index.html

Modify URI Translation

- Write a handler to modify the URI translate phase: PerlTransHandler

```
#file:MyApache2/RewriteURI.pm

package MyApache2::RewriteURI;

use strict;
use warnings;
use Apache2::RequestRec ();
use Apache2::Const -compile => qw(DECLINED);

sub handler {
 my $r = shift;
 my ($date, $id, $page) = $r->uri =~ m|^/news/(\d+)/(\d
+)/(.*)|;
 $r->uri("/perl/news.pl");
 $r->args("date=$date;id=$id;page=$page");
 return Apache2::Const::DECLINED;
}
1;
```

- Grab the parts of the old URI by calling the uri method
- Reset the URI using the uri method as a setter
- Set the proper query string using the args method
- Return Apache2::Const::DECLINED to allow any other translation handler to run

Filters

Filters

- Apache 2 added filters, allowing you to process incoming and outgoing data
- Independent of the request cycle processing
- mod_perl 2 gives full access to filtering

Filters

- Two types of filters
  - HTTP request filters; operate on the request body only
  - Connection filters; operate on the headers and body

Simple Example

- Output filter to remove line endings from all HTML output
- HTTP request filter, so body only
- Uses Apache2::Filter streaming interface to access request data

```
sub handler {
  my $f = shift; # Apache2::Filter object

unless ($f->ctx) { # Only run first time
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1); # Store flag in filter context
  }

 while ($f->read(my $buffer, BUFF_LEN)) {
    $buffer =~ s/[\r\n]//g;
    $f->print($buffer);
  }
return Apache2::Const::OK;
}
```

- Get an Apache2::Filter object as first argument
- Unset Content-Length since we will be modifying the size of the document
- Use the read method to grab BUFF_LEN of data
- Modify it
- Use the print method to send the data to the next filter in the chain

```
# Apache configuration
<Files ~ "\.html">
PerlOutputFilterHandler Apache2::Filter
</Files>
```

Simple Example

- More complex examples work directly with Apache data buckets
- More involved, but there are no limitations
- See Apache::Clean for a full version of the HTML cleaner

Handler Basics

- Quick overview of what's possible
- Many details depend on what you are doing
- Also relies on Apache knowledge
- Set up a good test environment and experiment
- Start small and build up complexity

Questions

Migrating to mod_perl 2

Porting your mod_perl 1 code

Why Change?

- You need mod_perl 2 to run with Apache 2
- Apache 2 is the current generation of Apache
- Support, new features, active development will start trailing off for Apache 1 (and likely mod_perl 1)

Bad News

- Your mod_perl 1 code won't run under mod_perl 2
- Many things moved because the Apache API changed
- Many other things moved to a more sensible location

Don’t Panic

Quick Fix

- If you need a quick fix, use Apache2::compat
- Instant compatibility layer
- Not intended for CPAN module use

Quick Fix

- Personal dev server
- Want to use a module like Apache::Pod::HTML right away
- (Actually, it's already been migrated to Apache2::Pod::HTML.)

Apache2::compat

- Loads all the mod_perl 2 Apache2::* modules
- Adjusts method calls where the prototype has changed
- Provides a Perl implementation for methods that no longer exist

Not a Final Solution

- Can cause collisions with mod_perl 2 methods with the same name (see docs for override feature)
- Loads everything, so it can have a large memory impact
- Can be slower because some compat code is Perl rather than C

Not 100%

- May not work for all modules because of other changes.
- Apache::Debug, content_language method was supposed to be content_languages
- Dropped because it was incorrect, so not available via compat

Other Options

- Modify the module to run under mod_perl 2
- Modify the module to run under both mod_perl 1 and 2

Example Migration

Apache::Debug

Initial Steps

- Get module running with Apache2::compat
- Should weed out incompatible code
- Remove Apache2::compat

Code-specific fixes

- The fixes at this point will likely be specific to your code
- Not covered by Apache2::compat
- Likely not fully documented
- Web and mailing-list searches reveal many solutions

Change Summary

- content_language now content_languages, takes array ref
- write_client changed to print
- remote_logname in the connection object is now get_remote_logname in $r (via Apache2::Access)
- headers_in now returns an APR::Table object

Final Migration

Remove Apache2::compat

- Next step is to remove the compatibility layer
- Make all the changes Apache2::compat was providing
- Make sure it's really gone and not pulled in by something else

Porting Tools

- Apache2::porting
  - Traps non-mod_perl 2 method calls
  - Suggests new locations for methods and modules

Process

- In httpd.conf, load:
  - Apache2::Reload
  - Apache2::porting
- Watch the error log and fix errors using the suggestions

Resources

- Error log can offer some direction
- Migration reference (User's Guide)
- mod_perl web site
- Mailing lists (often found via web searches)
- Google searches

- use many modules (error log)
- get_remote_host in Connection obj (error log)
- headers_out now an APR::Table object (mod_perl docs)
- send_http_header no longer needed (User's Guide)
- user now in $r (error log)

- auth_type now ap_auth_type in $r (error_log)
- For server_admin, add use Apache2::ServerRec (); (error_log)
- $r->args and $r->content now different, need Apache2::Request (Migration reference)

Final Cleanup

- Only run under mod_perl 2:

  use mod_perl 2.0;

- Change namespace, if needed (maybe Apache2::)

More Tools

Code Loaded on Startup

- Apache2::porting is limited to code loaded on child creation
- Use ModPerl::MethodLookup for other code
- Provided with mod_perl 2

```
>perl -MModPerl::MethodLookup -e
print_method server_admin

To use method 'server_admin' add:    use
Apache2::ServerRec ();
```

Set an alias

- To make it easier:

```
> alias lookup "perl -
MModPerl::MethodLookup -e
print_method"

> lookup server_admin
```

MethodLookup in code

- Some methods exist in more than one module
- lookup_method function accepts an object to help find the correct module
- ModPerl::MethodLookup:: lookup_method('print', $r);

Loading all modules

- You can load all modules at once with the function ModPerl::MethodLookup::preload_all_modules
- Not recommended for production
- See ModPerl::MethodLookup docs for more methods

Running under mod_perl 1 and 2

Running under 1 and 2

- Detect which version is running:

```
use constant MP2 =>
($mod_perl::VERSION >= 2.0);

use constant MP_GEN =>
$ENV{MOD_PERL} ? eval { require
mod_perl; $mod_perl::VERSION >=
2.0 ? 2 : 1 }
```

Load the correct modules

```
BEGIN {
 if (MP2) {
  require Apache2::RequestRec;
 }
 else{
  require Apache;
 }
}
```

```
BEGIN {
 if (MP2) {
  require Apache2::Const;
  Apache2::Const->import(-compile => qw(OK
DECLINED)); }
 else {
  require Apache::Constants;
  Apache::Constants->import(qw(OK
DECLINED));
}}
sub handler {
 # Do something
 return MP2 ? Apache2::Const::OK :
Apache::Constants::OK;
}
```

Make calls based on version

```
$r->send_http_header() unless MP2;
```

Method Handlers

```
sub handler_mp1 ($$) { &run }

sub handler_mp2 : method { &run }

*handler = MP2 ? \&handler_mp2 :
\&handler_mp1;

sub run { ...
```

Add Tests

- Good opportunity to implement Apache::Test
- Create baseline tests with your mpl code
- Keep running them as you migrate

Questions