PHP Performance Profiling

Jonathan Oxer

Contents

Background

With the incredible growth of PHP[1] in the last couple of years it's now being used for a huge range of tasks ranging from tiny scripts to large-scale web applications. Some web applications have hundreds of thousands of lines of PHP code, and the fact that PHP can scale to these sorts of levels is a great testament to its design and the efficient Zend Engine[2] that actually manages PHP code execution.

Of course, bigger and more complex projects result in more load on your servers, and when you throw a database into the mix you have even more potential performance bottlenecks to keep track of. A typical scenario is that you've added a few new features to a web application, and now you're seeing more server load and memory usage, and pages seem to load slower. What can you do? Maybe you can afford to throw bigger hardware at the problem, but even if that's a viable option you should also find the parts of your code that are causing slowdowns and optimize them.

There are a number of factors that can affect the performance of a web application, including:

- Web server configuration
- Database performance
- Data structure
- The application design
- Implementation of the application

I'm going to assume that you've already taken care of the first items (you have, right?) and now you're wanting to iron out bottlenecks in your application implementation: that is, your actual code. But how do you find the bottlenecks in the first place?

The answer is a technique known as performance profiling.

Performance profiling runs your code in a controlled environment, and returns a report giving statistics such as time spent within each function, how long each database query takes, and how much memory has been used.

By doing performance profiling on your code you will very quickly see where you may be wasting time with slow database queries or inefficient code, allowing you to then spend your time tuning your PHP and SQL in the places it needs it most. No more guessing what's going on internally: performance profiling will give you hard figures!

Profiling Tools

A number of different tools have been developed to help with PHP performance profiling, including:

- Benchmark (a PEAR project)[3]
- DBG[4]
- Xdebug[5]
- Advanced PHP Debugger (another PEAR project)[6]

If you're really serious about squeezing every last cycle out of your code you should probably

investigate all the benchmarking tools you can find, since they work in different ways and allow you to extract different kinds of information. However, for now I'm going to concentrate on APD: the Advanced PHP Debugger.

APD is a debugger written in C by George Schlossnagle and Daniel Cowgill that loads as an extension to the Zend Engine. It works by hooking into the Zend internals and intercepting PHP function calls, allowing it to do things like measure function execution time, count function calls, perform stack backtraces, and other funky things.

Installing APD

There are currently three main ways to install APD on a Linux system: grab the source and compile it yourself, use PEAR, or use the Debian package.

The latest source is always available from the APD website at pear.php.net/apd. Building and installing it isn't very hard, but you'll need to make sure you have the various PHP development resources installed on your system. For example, you'll need the PHP C headers, as well as a program called phpize that is used to prepare the package as a Zend extension. If you decide to go down that route, make sure you follow the instructions in the README included with the source.

If you use PEAR, included with PHP4.3+, you can install PHP modules with minimal fuss. Once again, the full instructions are on the APD website which is part of the PEAR project. Assuming you have PEAR support in your version of PHP, getting it going should be as simple as typing 'pear install apd' and answering a few questions.

Finally, for Debian users I maintain a .deb package of the latest version of APD. It's much too recent to be in Woody, but at the time of writing it's in Sid (current Unstable) and should enter Sarge (current Testing) very soon. You should be able to just do 'apt-get install php4-apd' and everything will be done for you.

Note that whatever installation method you use, you should have the CGI version of PHP installed because some of the command-line tools included with APD are written in PHP and need the parser to run. Personally I run my web servers with the Apache-module version of PHP because it's much faster, but that doesn't matter: just install the CGI version of PHP as well and away you go. It doesn't need to affect your Apache-module installation of PHP, they can live side by side quite happily.

You can use the phpinfo() function to check that APD installed and loaded properly. Just create a file in your web root that calls phpinfo(), and open it in a browser. The quickest way to do that is probably just to type

```
echo '<?php phpinfo() ?>' > info.php
```

in your web root. When you access the file in your browser through your web server (not by directly opening the file!) it'll tell you all the extensions that PHP has loaded, and you should see 'APD' listed somewhere down the page. If that worked fine, you're ready for the next step.

Your First Test

At this point you should have APD successfully installed on your server, along with all your

usual stuff – Apache, MySQL or Postgres, your PHP scripts themselves, and whatever else you need to run your web application.

Now pick a PHP script in your web app for a test run, and open it in an editor. In my example here I'm going to use the file 'src/webmail.php' from Squirrelmail[7], a popular webmail system written in PHP.

The basic procedure is to invoke special APD functions inside your PHP script which tell APD to do its stuff. So right at the top of your PHP script, put in a call to 'apd_set_pprof_trace()' like this:

```
<?php
apd_set_pprof_trace();
...etc
```

What that does is tell APD to start a trace on execution of your script at that point, and dump it out to disk in a predefined location which is set in the php.ini file using the 'apd.dumpdir' directive. If you used my Debian package the location will be set to /var/log/php4-apd/, if you used PEAR or compiled from source please check the included documentation.

Gathering Some Data

Now that your script is all set up and ready to profile, load it in a web browser. It should run exactly as before: you shouldn't see any difference at all from the client side, the page will load as it always has.

What will be different this time is that a pprof tracefile will have been written out to the dumpdir defined previously. A pprof tracefile is a text file containing a machine-parsable summary of how your PHP was processed, and will be named something like 'pprof.25802'. The number is just the process ID of the web server process that handled the request. You can take a quick look through the tracefile but at this point it won't mean much to you. For my example using Squirrelmail it looks something like this:

```
#Pprof [APD] v0.9
hz=100
caller=/jade/webserver/site13/web/squirrelmail-1.2.5/src/webmail.php

END_HEADER
! 1 /jade/webserver/site13/web/squirrelmail-1.2.5/src/webmail.php
& 1 main 2
+ 1 1 2
- 2 98816
! 2 /jade/webserver/site13/web/squirrelmail-1.2.5/functions/strings.php
& 3 require_once 2
+ 3 2 16
& 4 php_self 2
+ 4 2 597
@ 1 0 1
- 4 244352
- 3 244368
! 3 /jade/webserver/site13/web/squirrelmail-1.2.5/config/config.php
+ 3 3 17
- 3 291224
... etc
```

... and so on for several more pages. Essentially it's a step by step record of what the Zend engine did as it processed your script, but the format is not designed for direct human consumption: it's an intermediate log that can then be processed to generate nice reports.

Interpreting The pprof Tracefile

APD comes with a little command-line script written in PHP called 'pprofp' which can be run from the command line to parse the pprof tracefile and give you a human readable report. To use it, run pprofp and pass it a flag and the path of the tracefile like this:

```
pprofp -u /var/log/php-apd/pprof.25802
```

It will then print out a function call summary somewhat like this:

```
Trace for /jade/webserver/site13/web/squirrelmail-1.2.5/src/webmail.php
Total Elapsed Time =    0.15
Total System Time  =    0.02
Total User Time    =    0.13


         Real         User        System             secs/    cumm
%Time (excl/cumm)  (excl/cumm) (excl/cumm) Calls    call    s/call  Memory Usage Name
--------------------------------------------------------------------------------------
 30.8  0.05  0.12   0.04  0.11   0.01  0.01    22   0.0018    0.0050       859752 require_once
 15.4  0.02  0.02   0.02  0.02   0.00  0.00     8   0.0025    0.0025       403200
register_attachment_common
 15.4  0.02  0.02   0.02  0.02   0.00  0.00    23   0.0009    0.0009       298360 define
  7.7  0.01  0.01   0.01  0.01   0.00  0.00     1   0.0100    0.0100       145536 php_self
  7.7  0.01  0.01   0.01  0.01   0.00  0.00     7   0.0014    0.0014       342368
function_exists
  7.7  0.02  0.02   0.01  0.01   0.01  0.01     1   0.0100    0.0100       457224
session_start
  7.7  0.01  0.01   0.01  0.01   0.00  0.00     4   0.0025    0.0025          968
cacheprefvalues
  7.7  0.01  0.01   0.01  0.01   0.00  0.00     1   0.0100    0.0100       163952 is_array
  0.0  0.00  0.01   0.00  0.01   0.00  0.00     4   0.0000    0.0025           64 getpref
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           16 is_logged_in
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000         -248 ereg
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           16
set_up_language
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           72 ini_get
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000          232 setlocale
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000          256 header
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     3   0.0000    0.0000          296 putenv
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           96 getenv
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           56 textdomain
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000          424
bindtextdomain
  0.0  0.00  0.00   0.00  0.00   0.00  0.00     1   0.0000    0.0000           56 do_hook
```

Just a little easier to understand than the raw trace file!

The report above shows usage of time and memory on a per-function basis, sorted by user-time by the '-u' switch. The first few columns are just execution time in seconds. The 'Calls' column is a count of the number of times that function was executed by the script. 'secs/call' is the average execution time of each call to that function, while 'cumm s/call' is the cumulative time spent in that function. Then it lists memory usage and finally the name of the function itself. Note that function call reports are truncated to 15 functions by default.

If your script has some major performance problems, this is about the time they should start to become glaringly obvious. Do you have a slow function that's called many times? Better take a close look at it. Are you doing lots of SQL queries? You'll see them here right away.

pprofp Options

The basic report shown above is very useful for getting an overview of execution of your script, but you can also pass other switches to pprofp to tell it to format its report in different ways. Just call it using 'pprofp <switch> <tracefile>' using one or more of these options:

```
Sort options
-a          Sort by alphabetic names of subroutines.
```

```
-l           Sort by number of calls to subroutines
-m           Sort by memory used in a function call.
-r           Sort by real time spent in subroutines.
-R           Sort by real time spent in subroutines (inclusive of child calls).
-s           Sort by system time spent in subroutines.
-S           Sort by system time spent in subroutines (inclusive of child calls).
-u           Sort by user time spent in subroutines.
-U           Sort by user time spent in subroutines (inclusive of child calls).
-v           Sort by average amount of time spent in subroutines.
-z           Sort by user+system time spent in subroutines. (default)

Display options
-c           Display Real time elapsed alongside call tree.
-i           Suppress reporting for php builtin functions
-O <cnt>     Specifies maximum number of subroutines to display. (default 15)
-t           Display compressed call tree.
-T           Display uncompressed call tree.
```

If you've got a lot of functions (subroutines) in your script, it may be helpful to sort by number of calls (pprofp -l <tracefile>), or memory used (pprofp -m <tracefile>) to quickly see where the bottlenecks are.

Function Call Tree

While the function call report is probably the most immediately useful for finding bottlenecks, one of the funkiest options in the pprofp script is the ability to output a function call tree. A function call tree is essentially a step by step list of each function that was executed, indented to show function nesting.

You can output a function call tree with the -t or -T switches, like this:

```
pprofp -t /var/log/php4-apd/pprof.15507
```

which will output something like this for my Squirrelmail example:

```
main
require_once
  php_self
require_once (2x)
  session_is_registered
  require_once
require_once
  require_once (3x)
    require_once (2x)
  require_once
    require_once
      require_once
        require_once
          is_array
          use_plugin
            file_exists
            include_once
            function_exists
            ... etc
```

As you can see, a 'require_once' was performed and inside that a 'php_self' was executed. Then another 'require_once' executed 'session_is_registered', followed by another 'require_once', and so on.

Basically the function call tree is like a little window into the Zend Engine, allowing you to watch the sequence of events that take place when your web app is run.

If you do much object-oriented development you'll find you rapidly lose track of what's actually going on inside some classes and methods. It's tempting to think of classes as a 'black box' because that's how we're taught to use them, but when optimising a complex web app you need to know what's actually going on inside each one or you may have performance bottlenecks you're not even aware of.

APD Function Reference

APD provides quite a number of functions that you can use to help profile and debug your code. Experiment with these to really see the gory internal details of what the Zend Engine is doing with your script, but note that some of them are now strongly deprecated for PHP4.3+:

apd_set_pprof_trace()

The most useful APD function as far as profiling is concerned, this dumps a tracefile named 'pprof.<pid>' in your apd.dumpdir. The tracefile is a machine parsable output file that can then be processed with the pprofp <tracefile> command.

apd_set_session_trace(N)

Similar to apd_set_pprof_trace(), it dumps a human-readable session trace named 'apd_dump_<pid>' in your apd.dumpdir. This is the old way of doing things, noted here just because it still works (for now). It's been deprecated, so YMMV. It's better to use a pprof trace instead. 'N' is an integer that sets what items will be traced: just use a value of 99 for now to turn on all implemented options.

array apd_callstack()

Returns the current call stack at that stage of execution as an array. Very cool!

apd_cluck([string warning[,string line delimiter])

Behaves like perl's Carp::cluck. Throw a warning and a callstack. The default line delimiter is '
\n'. This function is deprecated for users of PHP4.3+: use the internal debug_backtrace() and debug_print_backtrace()instead.

apd_croak([string error[, string line delimiter]])

Behaves like perl's Carp::croak. Throw an error, a callstack and then exit. The default line delimiter is '
\n'. This function is deprecated for users of PHP4.3+: use the internal debug_backtrace() and debug_print_back-trace() instead.

array apd_dump_regular_resources()

Return all current regular resources as an array.

array apd_dump_persistent_resources()

Return all persistent resources as an array.

override_function(string func_name, string func_args, string func_code)

Syntax similar to create_function(). Overrides built-in functions (replaces them in the symbol table).

rename_function(string orig_name, string new_name)

Renames orig_name to new_name in the global function_table. Very useful for temporarily overriding built-in functions.

Profiling A Live Site

To get the best understanding of how your PHP runs under real conditions, you might want to do your profiling in an environment that is as close as possible to the real thing. Real data and many real or simulated client connections are helpful. You can do that by making a duplicate of your site and running HTTP benchmarking software or even just a couple of shell scripts and Wget to simulate user load, or by profiling right on your live site.

But as you've seen, using the profiling features of APD in your scripts causes the web server to write out text files for every script access. That's obviously A Very Bad Thing if you're trying to profile a live site, because you'll have data dumped out faster than you can deal with it. Besides, it'll cause a slowdown on the server if your site has a lot of visitors (and if it didn't, you wouldn't be profiling it, would you!). A useful little trick to get around the problem is to only activate profiling for page requests coming from your specific IP address, allowing you to browse the live site and produce profile information but without any action being taken for requests by other users.

It's actually pretty trivial to implement. Just wrap the invocation of apd_set_pprof_trace() in a check for the requesting IP address, like this, and then only your nominated client addresses will cause the script to go into profiling mode:

```
<?php
$DEBUGIPS = array('203.222.90.204','192.168.0.23');
if(array_search($_SERVER[REMOTE_IP], $DEBUGIPS)) {
          apd_set_pprof_trace();
}
?>
```

Editor Integration

While code profiling is an extremely powerful technique it's not used nearly as often as it should be. Other than lack of awareness, probably the major factor preventing developers use it is the complexity of running a trace or extracting a call tree.

Ideally profiling tools like APD need to be tightly integrated into development environments such as text editors. At present APD still needs to be run manually, but recently I've been in talks with Andrew Jeffriess, author of gPHPEdit[8], about integrating it directly into gPHPEdit to allow one-click generation of profiling reports from PHP files directly as they're being edited.

Where To From Here?

What now? Optimize your code, of course! I'm not going to tell you how to do that here because it's way beyond the scope of this short paper, but at least now you'll have a good insight into the internal workings of your app and won't be wasting time optimizing parts of your code that aren't really a problem. And when you do make changes, you'll have a benchmark against which you can test your code to see how the changes affect performance and efficiency.

[1] www.php.net
[2] www.zend.com
[3] pear.php.net/benchmark/

[4] dd.cron.ru/dbg/
[5] xdebug.derickrethans.nl
[6] pear.php.net/apd/
[7] www.squirrelmail.org
[8] www.gphpedit.org