

PHP Performance Profiling

*Jonathan Oxer <jon@oxer.com.au>
Open Source Developers Conference
Melbourne, Australia
December 1st, 2004*

Due to the incredible growth of PHP in the last couple of years it's now being used for tasks ranging from tiny scripts to large-scale Web applications. Some Web applications contain hundreds of thousands of lines of PHP code, and the fact that PHP can scale to these levels is a great testament to its design and the efficient Zend Engine 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 track. A typical scenario might be you've added a few new features to a Web application and now are seeing more server load and memory usage; thus, 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 also should find the parts of your code that are causing the slowdowns and optimize them. A number of factors can affect the performance of a Web application, including Web server configuration, database performance, data structure, the application design and the implementation of the application. I'm going to assume you've already taken care of the first items, and now you want to iron out bottlenecks in your application implementation, that is, in the 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 listing such statistics as time spent within each function, how long each database query takes and how much memory has been used. If you do much object-oriented development you may find that 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 optimizing a complex Web application you need to know what's actually going on inside each object or you may have performance bottlenecks you don't even notice.

By doing performance profiling on your code, you quickly can see where you may be wasting time with slow database queries or inefficient code. Having this information then allows you to spend your time tuning PHP and SQL where it needs it most. No more guessing what's going on internally: performance profiling gives you hard figures.

Profiling Tools

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

- Benchmark (a PEAR project)
- DBG
- Xdebug
- Advanced PHP Debugger (another PEAR project)

If you're really serious about squeezing every last cycle out of the code you should investigate all the benchmarking tools you can find, because they work in different ways and allow you to extract different kinds of information. For now, however, 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

Currently three main ways exist to install APD on a Linux system: grab the source and compile it yourself, use PEAR or use the Debian package. The latest source always is available from the APD Web site. Building and installing it isn't a hard process, but you need to make sure the various PHP development resources are installed on your system. For example, you 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 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 available on the APD Web site, which is part of the PEAR project. Assuming PEAR support is included 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 this writing it's in Sid (current Unstable) and should enter Sarge (current Testing) soon. You should be able to use `'apt-get install php4-apd'` to have everything done for you.

Whatever installation method you use you should have the CGI or CLI 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. Simply 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 confirm that APD installed and loaded properly. Create a file in your web root that calls `phpinfo()`, and open it in a browser. A quick way to do that is 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 lists all the extensions that PHP has loaded. You should see APD listed somewhere on the page. If that worked fine, you're ready for the next step.

Your First Test

At this point you should have APD installed on your server, along with all the other usual stuff - Apache, MySQL or PostgreSQL, your PHP scripts themselves, and whatever else you need to run your web app.

Now pick a PHP script in your app for a test run and open it in an editor. In my example here, I use the file `src/webmail.php` from Squirrelmail, a popular Web mail system written in PHP. The basic procedure is to invoke special APD functions inside your PHP script that tell APD to do its stuff. 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 this 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 should 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 set up and ready to profile, load it in a web browser. It should run exactly as before, and you shouldn't see any difference at all from the client side; the page loads as it always has.

What is different this time is a `pprof` tracefile has been written out to the `dumpdir` previously defined. A `pprof` tracefile is a text file that contains a machine-parsable summary of how your PHP was processed, and it is named something like `pprof.25802`. The number is 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. Instead, it's an intermediate log that can be processed to generate nice reports.

Interpreting The `pprof` Tracefile

APD comes with a little shell script written in PHP called `pprofp` that 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 an option and the path of the tracefile, like this:

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

This command prints out a function call summary that should look something 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
```

%Time (excl/cumm)	Real (excl/cumm)	User (excl/cumm)	System (excl/cumm)	Calls	secs/ call	cumm 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

The report above shows time and memory usage on a per-function basis, sorted by user-time as directed by the `-u` option. The first few columns are 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 on that function. Then it lists memory usage and finally the name of the function itself. Notice that function call reports are truncated to 15 functions by default.

If your script has some major performance problems, here is where 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 can see them here right away.

pprofp Options

The basic report shown above is useful for getting an overview of the script's execution, but you also can pass other options to `pprofp` that tell it to format the report in different ways. Simply call it using

```
pprofp <option> <tracefile>
```

using one or more of the sort 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 built-in functions.

```
-O <cnt>: specify maximum number of subroutines to display. (default 15)
-t: display compressed call tree.
-T: display uncompressed call tree.
```

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

Function Call Tree

Although the function call report probably is the most immediately useful one 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 executed function, with indentation to show function nesting.

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

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

which outputs 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
        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 application is run.

One technique I've found particularly helpful in finding slow sections of code is to output a call tree with the real elapsed time option and then walk through it with 'less', ie: 'pprofp -Tc <tracefile> | less'. You'll then get a result that looks something like this:

```
jon@svn.it.com.au: /home/jon
0.87      dbase_sql->fetch_array
0.87      mysql_fetch_array
0.87      dbase_sql->fetch_array
0.87      mysql_fetch_array
0.87      dbase_sql->fetch_array
0.87      mysql_fetch_array
0.87      dbase_sql->fetch_array
0.87      mysql_fetch_array
0.87      dbase_sql->fetch_array
0.87      mysql_fetch_array
0.87      dbase_sql->fetch_array
0.87      mysql_fetch_array
0.87      contact->staff_array
0.87      strlen
0.87      dbase_sql->query
0.87      mysql_query
1.08      mysql_errno
1.08      mysql_error
1.08      dbase_sql->fetch_array
1.08      mysql_fetch_array
1.08      knowledge->article_display
1.08      ereg_replace
1.08      nl2br
1.08      eval
1.08      ob_get_contents
1.08      ob_end_clean
```

The trick is to scroll down the call tree looking for large time increments. In this example you can see there's a time jump of 210 milliseconds when `mysql_query` is called, which indicates the query is costing a lot of time. That's a clue that you need to examine the query and determine if you can make it faster by simplifying the data structure or even just the query itself.

Then you can re-run the script and measure whether you've actually made a tangible improvement, rather than just guessing or trying to determine if the app 'feels' faster.

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 see the gory internal details of what the Zend Engine is doing with your script, but note that some of them now are 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.dumppdir`. The tracefile is a machine-parsable output file that can 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.dumppdir`. This is the old way of doing things, noted here because it still works (for now). It's been deprecated, so it's better to use a `pprof` trace instead. `N` is an integer that sets the items to be traced; for now, use a value of 99 to turn on all implemented options.

array apd_callstack()

Returns the current call stack at that stage of execution as an array. `apd_cluck([string warning [,string line delimiter]])` Behaves like Perl's `Carp::cluck` module. Throws 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 module. Throws an error, a callstack and then exits. 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.

array apd_dump_regular_resources()

Returns all current regular resources as an array. `array apd_dump_persistent_resources()` Returns all persistent resources as an array. `override_function(string func_name, string func_args, string func_code)` Syntax is 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`. Useful for temporarily overriding built-in functions.

Profiling A Live Site

To get the best understanding of how your PHP implementation runs under real conditions you should do the profiling in an environment as close as possible to the real thing. Real data is vital since many performance problems are a result of data structures that don't scale well, and many real or simulated client connections may also show you some weaknesses. You can create this environment by making a duplicate of your real site and running HTTP benchmarking software or even using shell scripts and Wget to simulate user loads. Alternatively you can do the profiling right on your live site.

As you've seen, using the profiling features of APD in your scripts causes the Web server to write out log files for every script access. Those log files can be fairly large and it's obviously a very bad thing if you're trying to profile a live site, because data is dumped out faster than you can deal with it. Besides, it causes 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 activate profiling only for page requests coming from your specific IP address. Doing so allows you to browse the live site and produce profile information without any action being taken on requests by other users.

The solution is actually pretty trivial to implement. Simply wrap the invocation of `apd_set_pprof_trace()` in a check for the requesting IP address as shown below. Then only your nominated client addresses cause the script to go into profiling mode:

```
<?php
$DEBUGIPS = array('203.222.90.204', '192.168.0.23');
if(array_search(getenv("REMOTE_ADDR"), $DEBUGIPS)) {
    apd_set_pprof_trace();
}
?>
```

Extreme Funkiness: APD And Kcachegrind

Kcachegrind is a visualisation tool used to display the output of Calltree, a profiling skin for the Valgrind debugger for C code development. The CVS version of APD now includes a little script called `pprof3calltree`, which takes a `pprof` trace file and converts it to the Calltree format so Kcachegrind can read it.

That means you can now use Kcachegrind to visualise your PHP code!

At present pprof2calltree is not yet in the released version of APD so you will need to grab the CVS version. Start by putting this in your ~/.cvsrc file:

```
cvsv -z3
update -d -P
checkout -P
diff -u
```

Then log into the PHP CVS server with 'cvsread' as the username and 'phpfi' as the password:

```
cvsv -d :pserver:cvsread@cvs.php.net:/repository login
```

Next you need to check out the APD source code from the PECL tree:

```
cvsv -d :pserver:cvsread@cvs.php.net:/repository checkout pecl/apd
```

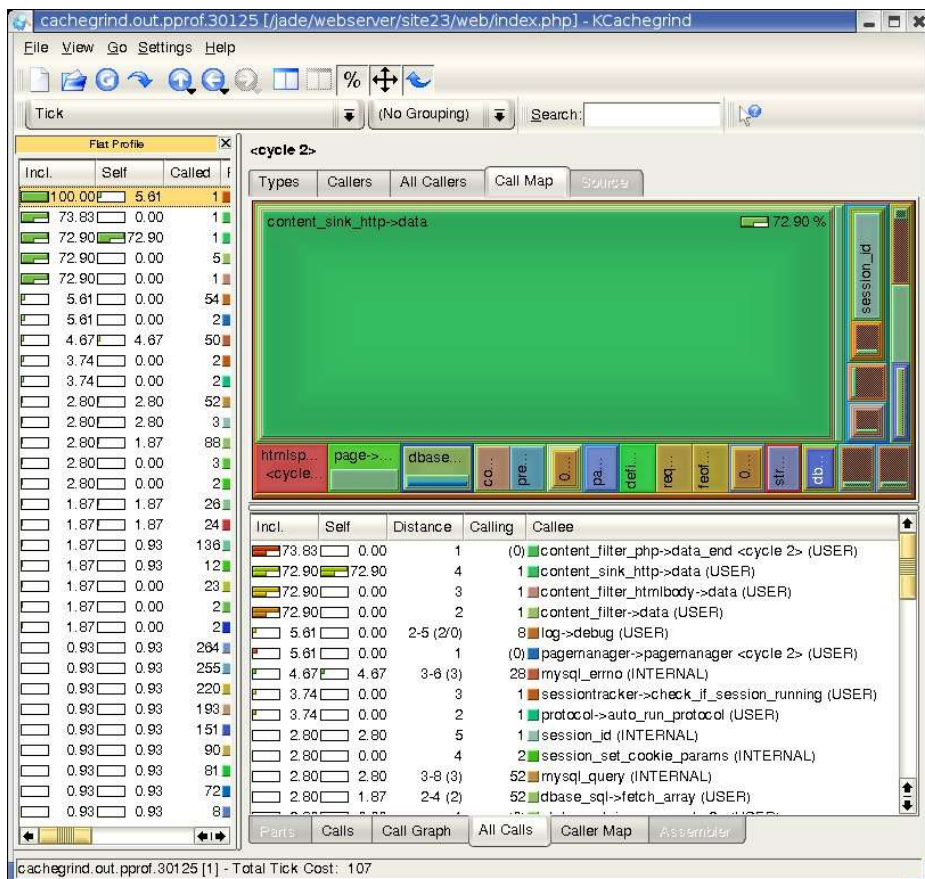
If you already have APD installed you don't need to re-install it, you can just use the 'pprof2calltree' script with your existing installation if you prefer. You also need to make sure you have Kcachegrind installed and operational, of course. So lets convert an existing pprof tracefile to calltree format:

```
./pecl/apd/pprof2calltree -f /var/log/php4-apd/pprofp.32061
```

Now you'll have a new file called 'cachegrind.out.pprof.32061' which Kcachegrind can read, so we can fire up Kcachegrind and take a look:

```
kcachegrind cachegrind.out.pprof.32061
```

And you'll get something funky like this:



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 paper, but at least you now have some insight into the internal workings of your application and won't waste time optimizing parts of your code that aren't really a problem. And when you do make changes, you have a benchmark against which you can test your code to see how the changes affect performance and efficiency.

About The Author

Jonathan Oxer is founder and Technical Director of Internet Vision Technologies, an Australian web application development agency with clients around the world. He is also a Debian developer, and organised Debian Miniconf 2 in Perth in January 2003 and Debian Miniconf 3 in Adelaide in January 2004 in association with Linux Conf Australia where he also presented technical papers. He has spoken on various Internet-related topics at both corporate and government seminars and conferences around the world, and is a regular guest on radio station RRR's "Byte Into It" segment. He sits on the Advisory Group of Swinburne's Centre for Collaborative Business Innovation. His first book, *How To Build A Web Website And Stay Sane* (www.stay-sane.com), will be published soon, and his second book, *The Debian Universe* (www.debianuniverse.com), is being written live on-line. He has been retained by O'Reilly to work on *Running Debian GNU/Linux*, and his fourth and fifth books (*Web-Based Business Intelligence* and *Disaster Proofing For Small Networks*) are already underway.

Jonathan can be reached at jon@ivt.com.au.