OPTIMIZING YOUR PHP WITH XDEBUG

I work with a lot of PHP applications, and part of my job is optimizing those applications to reduce server costs and maximize how many requests each server can handle. There are many ways to do this, but I'm going to talk about using a profiler, which is one of the better ways to start optimizing your code base.

You've probably heard people saying not to get caught up in premature optimization, maybe you've heard them follow up with profile your code to find the bottlenecks. A lot of people don't know what this means, or are under the impression that profiling is quite hard to setup. However, this is certainly not true. Profiling PHP is very simple, thanks to the wonderfulXdebug extension. Before we get started setting up Xdebug, you're going to need a program to read the profiling information files, known as cachegrind files. I personally recommend usingKCacheGrind (also available on Windows), although WinCacheGrind is another good alternative. Finally, there is a very lightweight profiler called WebGrind that is web based. You should download and install one of these programs.

Before continuing, a small note about callgrind/cachegrind files. Callgrind is a tool included with Valgrind, a C/C++ analysis tool. A lot of different profilers support this format, so the information about interpreting the results below are language agnostic.

Installing Xdebug

These instructions are for Ubuntu, however it’s pretty easy to find instructions for whatever platform you are on. Xdebug is quite popular.

Simply install the Xdebug PHP extension using apt, and you’re set:

```
sudo apt-get install php5-xdebug
```

You may need to restart Apache/PHP-FPM. You can verify it’s installed using phpinfo, or by running `php -i | grep xdebug`.

Configuring Xdebug

Next, we need to enable the Xdebug profiler (it’s disabled by default). We have two options here: Always on, or we can turn it on my setting a GET parameter or cookie. I recommend the second option, it’ll make your life easier.

Edit your php.ini file, and add the following option:

```
xdebug.profiler_output_dir = /path/to/store/cachegrind/files
```

I use the following:

```
xdebug.profiler_output_dir = /var/www/_profiler
```

Make sure the directory exists.

Next, enable the profiler.

Always On:

```
xdebug.profiler_enable = 1
```

Enabled using the GET/POST/COOKIE param XDEBUG_PROFILE=1:

```
xdebug.profiler_enable_trigger = 1
```

The above settings are mutually exclusive, so only use one of them.

Dumping A Cachegrind File

Cachegrind files contain all of the profiling information for your application.

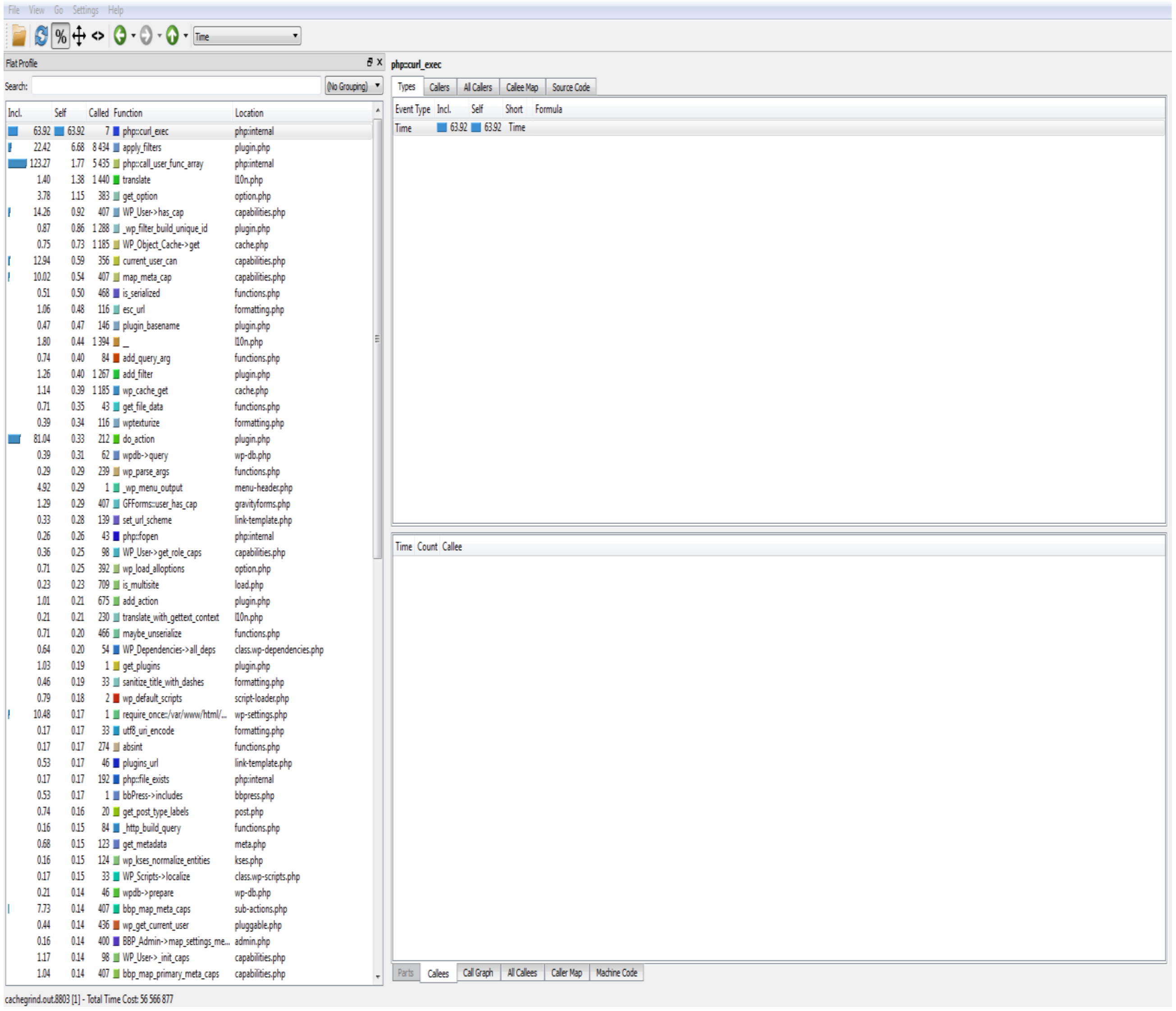
Simply navigate to your site with the GET parameter set, like so:

```
http://localhost:11000/?XDEBUG_PROFILE=1
```

In the configured profiler output directory, you should see a file named like this:

cachegrind.out.8803 (the number will be different for you)

Now, open this file using your cachegrind viewer.



Incl.	Self	Called	Function	Location
63.92	63.92	7	php::curl_exec	php:internal
22.42	6.68	8 434	apply_filters	plugin.php
123.27	1.77	5 435	php::call_user_func_array	php:internal
1.40	1.38	1 440	translate	l10n.php
3.78	1.15	383	get_option	option.php
14.26	0.92	407	WP_User->has_cap	capabilities.php
0.87	0.86	1 288	_wp_filter_build_unique_id	plugin.php
0.75	0.73	1 185	WP_Object_Cache->get	cache.php
12.94	0.59	356	current_user_can	capabilities.php
10.02	0.54	407	map_meta_cap	capabilities.php
0.51	0.50	468	is_serialized	functions.php
1.06	0.48	116	esc_url	formatting.php
0.47	0.47	146	plugin_basename	plugin.php
1.80	0.44	1 394	__	l10n.php
0.74	0.40	84	add_query_arg	functions.php
1.26	0.40	1 267	add_filter	plugin.php
1.14	0.39	1 185	wp_cache_get	cache.php
0.71	0.35	43	get_file_data	functions.php
0.39	0.34	116	wptexturize	formatting.php
81.04	0.33	212	do_action	plugin.php
0.39	0.31	62	wpdb->query	wp-db.php
0.29	0.29	239	wp_parse_args	functions.php
4.92	0.29	1	_wp_menu_output	menu-header.php
1.29	0.29	407	GFForms::user_has_cap	gravityforms.php
0.33	0.28	139	set_url_scheme	link-template.php
0.26	0.26	43	php::fopen	php:internal
0.36	0.25	98	WP_User->get_role_caps	capabilities.php
0.71	0.25	392	wp_load_alloptions	option.php
0.23	0.23	709	is_multisite	load.php
1.01	0.21	675	add_action	plugin.php
0.21	0.21	230	translate_with_gettext_context	l10n.php
0.71	0.20	466	maybe_unserialize	functions.php
0.64	0.20	54	WP_Dependencies->all_deps	class.wp-dependencies.php
1.03	0.19	1	get_plugins	plugin.php
0.46	0.19	33	sanitize_title_with_dashes	formatting.php
0.79	0.18	2	wp_default_scripts	script-loader.php
10.48	0.17	1	require_once::/var/www/html/...	wp-settings.php
0.17	0.17	33	utf8_uri_encode	formatting.php
0.17	0.17	274	absint	functions.php
0.53	0.17	46	plugins_url	link-template.php
0.17	0.17	192	php::file_exists	php:internal
0.53	0.17	1	bbPress->includes	bbpress.php
0.74	0.16	20	get_post_type_labels	post.php
0.16	0.15	84	_http_build_query	functions.php
0.68	0.15	123	get_metadata	meta.php
0.16	0.15	124	wp_kses_normalize_entities	kses.php
0.17	0.15	33	WP_Scripts->localize	class.wp-scripts.php
0.21	0.14	46	wpdb->prepare	wp-db.php
7.73	0.14	407	bbp_map_meta_caps	sub-actions.php
0.44	0.14	436	wp_get_current_user	pluggable.php
0.16	0.14	400	BBP_Admin->map_settings_me...	admin.php
1.17	0.14	98	WP_User->_init_caps	capabilities.php
1.04	0.14	407	bbp_map_primary_meta_caps	capabilities.php

Event Type	Incl.	Self	Short	Formula
Time	63.92	63.92	Time	

Time	Count	Callee

cachegrind.out.8803 [1] - Total Time Cost: 56 566 877

This is an example from a local development machine running WordPress, that was loading quite slowly for some reason. The first thing you should do, is sort by “Self” time, like I have. Please note that times here are relative, and don’t mean a whole lot by themselves.

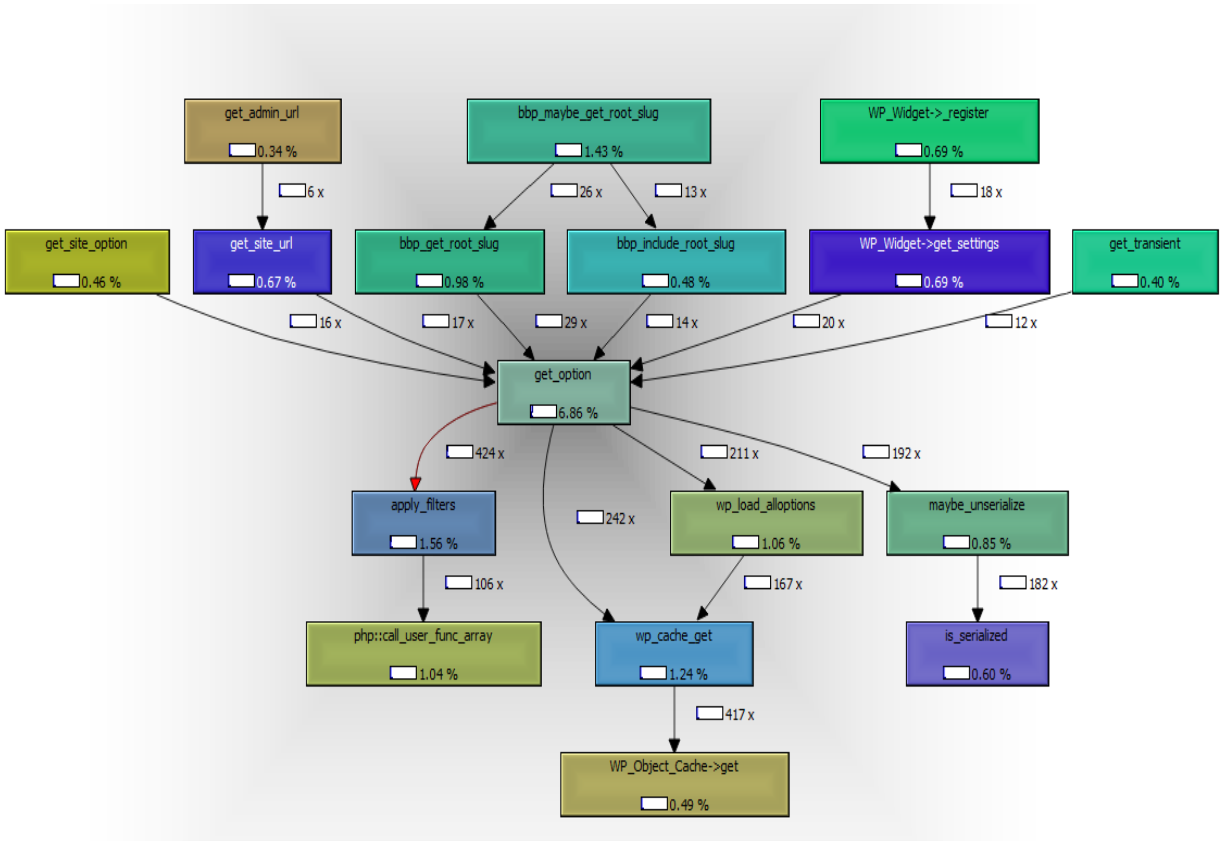
If you’re wondering what the different between “Incl” (Inclusive) and Self time is, it’s quite simple. Inclusive time is the time required to run a function including the time it took to run every other function called by this function. That’s why your main file will have the highest inclusive time, it runs everything. Self time is far more useful, and is the time it took to run just that function, not any functions called by it. That’s the column important for optimization. Here, we can see that curl_exec was called 7 times, and took about 10 times longer than the next slowest call.

You can use the callers tab to see which functions called each function, and step through the execution of your program. You can also browse the source code for the function being called (You may have to map directories to your source directory if you’re running kcachegrind on another machine, do this from the settings). I did this for the above curl_exec, and was able to narrow down the trouble to the WordPress version check, which I then disabled as I do updates to this site manually.

On another site, I noticed the custom routing function was taking a substantial amount of time to map routes. When I looked at the function, it was doing a bunch of work to clean up and normalize the routes. This was a developer convenience feature, but had a high performance cost. I removed all of the normalization code, and fixed all of the routes to ensure they were already normalized, and it sped the site up by about 12ms.

The Call Graph

The call graph, a tab located in the bottom right pane, is a useful visualization of the call graph for the currently selected function call.



A call graph is a directed graph that representing the calling relationships between functions in your code. The functions pointing TO get_option (in the middle) are functions that call get_option. The functions that get_option points to are functions called from in the get_option function itself. This tool is useful for visualizing the cost of a function and where the cost comes from.

Conclusion

Profiling your code allows you to find the slowest calls and spend effort fixing those, instead of blindly fixing things that may have very little impact on performance. It’s an essential tool for any developer. I urge any programmers reading this to get familiar with a profiling tool, whether it’s Xdebug for PHP or the equivalent for your language of choice. It will make you far more productive, and you’ll wonder how you lived without it!

Source: http://brandonwamboldt.ca/optimizing-your-php-with-xdebug-1455/