Scaling PHP Applications

By Steve Corona

Case Study: CakePHP Framework Caching

This is going to be a really awesome case study— I'm really pumped about it because it's an interesting problem that not only took an incredible amount of blood, sweat, and tears to solve, but is also extremely applicable to the theme of this book because it's something that could *only* be replicated at scale.

So first things first, I'll set the stage with the some background and context and walk you through how I debugged, found the core issue, failed at solving it several times, and finally found the perfect solution.

CakePHP, you devilish dessert

One of our several backend applications that I work on at Life360[133] is written in CakePHP. While I don't think CakePHP is awesome, it gets the job done well enough and is fairly easy to work with.

CakePHP is a *large* framework. It handles everything from autoloading your PHP files for you, to introspecting your database models and learning about the different columns, to providing a faux query cache to cut down on the number of `SELECT` queries you make to MySQL. Typical framework stuff.

Without some type of cache to remember all of this information about your code between requests, Cake would have to regenerate it for each pageload, adding a significant amount of overhead. In order to remember the data, Cake provides a pluggable system cache which can store this data using APC, flat files, Memcache, or MySQL.

While the exact mechanics are specific to CakePHP, the problem itself is generic— almost every substantial PHP and non-PHP framework does something similar, persisting framework and system cache data somewhere.

APC gets removed from PHP 5.4 (aka, "the core problem")

Our CakePHP framework cache was configured to use PHP's APC cache. APC works great for this type of workload because it's extremely fast and the data doesn't need to be shared between multiple servers. Sadly, APC has been removed in PHP 5.5 in favor of supporting the much more advanced (and now, open-source) Zend OpCache.

[133] http://www.life360.com

Remember that APC provides two different functions to PHP— it provides a userland cache, similar to Memcache, which is how our CakePHP framework cache is using it, but its primary function is to cache compiled PHP bytecode, allowing our code *execute* faster. The faux-memcache functionality is a secondary feature to its primary role as a bytecode cache.

Zend OpCache is better than APC— it benchmarks faster and is all around more modern. Hands down, it makes a ton of sense and it's why it has become the standard for PHP 5.5. Zend OpCache does one thing and does it well— it **only** does bytecode caching and doesn't offer a comparable userland cache like APC.

The reason I mention this is because when upgrading from PHP 5.4 to 5.5, we could no longer use APC for our CakePHP system cache. At first glance, it doesn't seem like it's that much data and it's not like it changes very much, so we just modified our configuration and told CakePHP to store its system cache data in flat files instead of APC— easy enough, right?

This worked seemingly well. No problems, it worked well at scale (400 million requests per day), everything was seemingly great. Little did I know, this small change would cause me to pull my hair out for months.

The life and death of a CakePHP request

Let's briefly run through how a PHP request is dispatched in CakePHP (or really, any MVC framework), and where the framework caching happens.

1. Web request is sent from the browser to Nginx, Nginx sends the request to PHP-FPM
2. PHP-FPM loads the bootstrap file of the framework, `index.php`
3. Bootstrap file loads framework configuration, **reads the framework cache data from the filesystem (if it doesn't exist, it has to regenerate it)**, and eventually invokes your controller.
4. Your controller code runs and generates a response for the client.
5. The framework **saves the newest system cache data to the filesystem** and sends your response upstream to nginx and eventually the browser.

There are two points at which the framework needs the cache data— at the beginning and end of the request. If the data doesn't exist, it has to regenerate it (figure out which classes map to which files, figure out which columns are in which tables, etc). Simple enough.

If it smells like a race condition...

So, with a fair understanding of how the cache works, what it accomplishes, and when it runs, let's move on to discussing where things started to fall apart.

It's typical Wednesday afternoon. We have some new code that we want to merge from our `development` branch to `master` and subsequently deploy to production. No big deal. Luckily, this

is pretty automated and as soon as `master` gets updated, `chef`[134] kicks off, updates the production servers with the latest code and reloads PHP-FPM.

But here's the weird part— when new code is deployed, we start seeing PHP-FPM processes pile up, quickly hit 100% CPU usage and spike server load to 80. We run out of free PHP-FPM children in no time, exceed the PHP-FPM backlog, and nginx starts throwing `502 Bad Gateway` HTTP responses. Bad news! And as quickly as it came, the problem would silently retreat and go away, never to be seen again until the next deploy.

It sort of became a ritual. Deploy new code, see a jump in errors, watch it mysteriously go away. It plagued me for months— no one wanted to deploy code, worried about causing brief downtime. We had a list of assumptions why this was happening (none of them were true), and it didn't go away until I sat down for 4 days straight and tested each one.

Still with me? Great. I want to walk you through this problem like we're working on the project together. I think the **process** it's what's helpful here— the answer itself isn't that valuable, that's why I'm walking you through the entire problem from start to finish instead of just giving you the solution.

My initial assumptions were that the increase in `502 Bad Gateway` errors were caused by one of the following problems.

1. Too many connections to Memcached due to reloading PHP
2. Reloading PHP-FPM (`kill -USR2`) wasn't working as expected
3. Zend OpCache taking time to warm up, causing many PHP requests to run without compiled bytecode and spiking CPU usage

Busting the assumptions

The first step to being able to test the three assumptions was to be able to replicate the problem. Unfortunately, it was nearly impossible to replicate this on a development server, because without a heavy flow of real traffic, code would deploy without issue. I segregated a production server that was serving traffic and force deployed to it over and over. Bingo— CPU spike, 502 errors, the works.

I can replicate the issue over and over again. Now I'm ready to start testing my assumptions, one by one.

Too many connections to Memcached from reloading PHP

My first assumption was that during the graceful reload of PHP, the old workers might still be holding onto their old connection to Memcached while the new ones start up. Memcached has a connection limit, if we exceed this, the memcached client will start throwing exceptions and these exceptions could be causing the problems we're experiencing.

[134] http://www.getchef.com/

Easy to test— if we force a deploy on our test box with 256 PHP-FPM workers, we should see the number of connections on our Memcached server go up by exactly 256. We can easily monitor this with `telnet`.

```
telnet memcache_server.ip 11211
> STATS
< STAT curr_connections 3840
...
```

Okay, we have 3840 connections to our memcache server, if we watch it while force deploying on our test box, we should see it blip up to 4096.

Deploy.. wait... and.. Nope. That wasn't it, after the deploy I see the number of connections hold steady at 3840. Myth busted.

Reloading PHP-FPM (`kill -USR2`) doesn't work as expected

When we deploy code, we gracefully reload PHP-FPM. Why reload PHP-FPM on deploy? We have `opcache.validate_timestamps=0` and `opcache.revalidate_freq=0` set in our `php.ini`. What these settings do is make it so that once Zend OpCache compiles the bytecode for a particular PHP file, it will never check the file for updates again. That is to say, if the file is modified or changed (manually or by a deploy), PHP will not read the latest updates until PHP-FPM is restarted or reloaded.

What does graceful reloading do? When you send the PHP-FPM master process the `USR2` signal, it waits for each child process to finish the current request and then replaces the old process with a new one. This allows PHP to gracefully restart without dropping any requests.

This one is easy to test. Instead of force deploying PHP, I just need to manually `kill -USR2` it on my test box and check for a spike in errors. So, I run:

```
$ kill -USR2 `cat /var/run/php-fpm5.pid`
```

And what do I see! A small jump in `502 Bad Request` errors. Is this it? Did we figure out our culprit? No, but it led me to discovering a small contributer. After reading the PHP-FPM source, I discovered that I needed to set `process_control_timeout` in my `fpm.conf` file, otherwise PHP-FPM will stall and wait forever for all of the children to finish their request. Setting `process_control_timeout=10s` made gracefully reloading PHP much smoother. But the main issue still persisted— a deploy, even with this change, still causes high CPU and 502 errors. Onwards.

Zend OpCache is taking a long time to warm up

The assumption here is that the OpCache takes a little while to warm up and that without the compiled PHP bytecode, the servers are running the PHP code slower and using much more CPU.

I was writing a bunch of logic in Chef to pre-warm the cache before deploying, I decided I should spend a few minutes testing this assumption before investing a ton of time trying to fix it.

I proved this assumption wrong in two ways:

The first (and easiest) way— I disabled Zend OpCache completely with `opcache.enable=0` in `php.ini` and reloaded PHP. While I saw that load was certainly higher, the server was not throwing `502 Bad Gateway` errors nor was the CPU at 100%. Busted.

The other way I tested was restarting PHP and after the first request printing the contents of `opcache_get_status()`. This function shows all of the files currently cached by Zend OpCache. It turns out that even with a very heavy framework like CakePHP, almost 75% of the code was compiled to bytecode after the first request.

Bummer! We're out of assumptions. Back to the drawing board.

Now what? I'm out of ideas

All of the assumptions that I had went down the tube as I systematically debunked each one. I was SURE that I knew what the problem was, but as it turns out, I was completely wrong. I'm out of ideas and what do I besides give up and pull out my hair?

`strace` to the rescue!

When you need a partner to help you debug, `strace` has got your back. It's one of my favorite tools on Linux and while you can spend an entire semester learning about all of the features it has, the simplest use-case can often be all that you need.

`strace` gives you the ability to attach to a running, live process (while marginally impacting performance— okay for our use, but don't `strace` a production MySQL server, for instance) and view what system calls it is making. Sure, you can't see the exact code that's executing, but you can see external network requests, filesystem calls, and other low-level stuff.

So, I deploy once again, and find myself a PHP-FPM worker to attach to:

```
$ ps aux | grep php-fpm

myapp 128131  2.9  0.0 324132 57396 ?        R    10:54  11:50 php-fpm: pool www
...
```

Perfect, quickly I attach to this process `128131` and start `strace`'ing it, to see what it's up to. Lo and behold, I see something interesting right away. The output flies by at mach-1, but sure enough it keeps pausing for several seconds after this block of system calls… it's stalling here!

```
open("/tmp/cake_core_file_map", O_RDWR|O_CREAT, 0666) = 10
fstat(10, {st_mode=S_IFREG|0666, st_size=29199, ...}) = 0
lseek(10, 0, SEEK_CUR)                  = 0
close(15)                               = 0
flock(10, LOCK_EX)                      = 0
```

So what the hell is this? Well, PHP is stalling at the last call to `flock`. What's `flock`? It's a system call to obtain a lock on the file, so no other programs can write to it. Why, it's waiting to get an exclusive file lock and blocking until it can get one. What file is it trying to obtain an exclusive lock on? OUR CAKEPHP SYSTEM CACHE FILES! Somewhere in the internal CakePHP code, it's locking the files before writing to them!

And that's the core problem. We found it. When we deploy a new code release, we're wiping out our `/tmp` directory— we WANT the CakePHP cache files to be refreshed after a deploy because a database table or model may change. But what's happening is that a stampede of 256 PHP-FPM workers are trying to update the cache simultanously, each requesting an exclusive file lock to do so. And that's where it's blocking— because of the competition over obtaining a file lock. The blocking causes requests to pile up, which causes our socket backlog to fill up, and forces Nginx to start throwing 502 Bad Gateway errors. That's it!

The solution(s)

Now that we know the issue, it's much easier to solve. We need to speed up `flock` (or avoid it all together) so that the during a deploy, we don't have 256 PHP-FPM workers all competing on trying to update the same files. There are a couple of different ways to do solve this problem, and I'll quickly cover which one I used (and the ones that failed).

Screw files, use Memcache!

My first reaction was to just use Memcache. Remember earlier we talked about the pluggable backend for the CakePHP system cache? Well one of the backends is Memcache, so screw it, let's just use that. I modified the configuration, plugged in my memcache server details, and it all seemed to work. But, curiousity got the best of me, and I wondered how much extra traffic this was adding to the Memcached server.

So I removed the change, and used `ifstat` to get a baseline for network traffic on this app server...

```
$ ifstat -i eth0
KB/s in  KB/s out
3154.15  2660.01
3003.02  2561.97
```

Ok, about 3MB/s in and out. Next, I re-enabled the memcached backend in CakePHP and checked again.

```
$ ifstat -i eth0
KB/s in  KB/s out
41931.34 33154.11
39129.12 31929.23
```

Oh wow, we added almost 40MB/s of traffic to and from our memcache server by using it for the CakePHP system cache. FOR ONE SERVER. Multiply it across all of our app servers and wow, it quickly becomes unsustainable. The CakePHP cache is only a couple of kilobytes but it needs to read it out and write back out the full contents for each request. Bummer, memcached isn't go to work for this.

APCu, the userland cache for PHP 5.5

Next up, I found APCu[135], the userland part of APC reimplemented for PHP 5.5. I tested this, and while it worked just fine, it felt dirty to me. This is a beta-level package, without much history or testing, and who knows if it will continously be updated to work with PHP 5.6, 5.7, and so on. Might as well make the clean cut instead of hacking the functionality back in with a legacy module. I will say this— out of all the solutions, this was probably the easiest and most straightforward. APC worked great for this solution in PHP 5.3 and 5.4.

An independent cache for each PHP-FPM worker

I had to wake up at 4:45AM on a Saturday morning to catch a flight to San Francisco. As I was flossing in my bathroom, it hit me in the face. Instead of relying on a *global* CakePHP system cache for all 256 of my PHP-FPM workers, why don't I have a unique file cache per-worker? This will get rid of the competition for the file lock, since each worker is single threaded and can only serve one web request at a time.

It gets even easier— it would only require one change:

```
<?php
// Old Configuration
Cache::config('_cake_core_', array('prefix' => 'cake_core'));

// New Configuration
Cache::config('_cake_core_', array('prefix' => getmypid() . 'cake_core'));
```

All we did was add `getmypid()` to the filename of the CakePHP framework cache— since each PHP-FPM worker runs as a child processes and has its own process id, each PHP-FPM worker gets its own unique cache.

[135] http://pecl.php.net/package/APCu

The best ideas come when you're flossing.

I was excited to land on the west coast and test my theory. Within minutes of landing, I tested my theory, and it worked. Except for a small problem. When I checked filesystem usage with `iostat`, I was saturating the disks! It was too much load on the disk to have 256 independent PHP-FPM caches! Blah!

```
$ iostat /dev/xvdb -x 1
Device:         rrqm/s   wrqm/s     r/s     w/s      svctm  %util
xvdb            1273.00  893.00     1263.00 893.00   1.21   100.00
```

Instead of going back to the drawing board AGAIN, I just iterated on this solution. The cache files were pretty small, less than 50KB each, so even with 256 independent caches we are only talking about 15MB or so— it's just getting read and re-written several thousand times per second.

I created an in-memory filesystem with `tmpfs` and mounted it in the location of my cache directory— giving CakePHP the *illusion* of using the filesysem, while really storing the cache in memory. This solved all of the problems and was ultimately the solution I used.

```
$ mount -t tmpfs -o size=1000M,mode=0755 tmpfs /tmp/cache
```

What a ride it was. I hope this case study showed you some real-world debugging, reasons to floss your teeth, and why at scale, sometimes even the obvious solutions don't always go as planned :)