July 14 HIPHOP · PHP · OPEN SOURCE

Under the hood: Box's HHVM migration

Erin Green

Every so often, we hear great stories about other companies adopting Facebook Open Source projects. Recently, the team at Box sent us this story of how they adopted HHVM. It's a good read — so we're posting it here. We thank them for sending it our way. We're always looking for feedback on our efforts. You can reach us on our ***Facebook Engineering page*** *or on* ***GitHub****.*

By Joe Marrama, software engineer, Box

Reducing latency and increasing the capacity of our infrastructure have always been top priorities at Box. We strive to deliver the best possible user experience in the most efficient manner, and historically our choice of PHP hasn't aligned well with these goals. I'm very happy to report that we've recently made very significant strides toward these two ideals by successfully deploying **HHVM** (the HipHop Virtual Machine) as the exclusive engine that serves our PHP codebase. In the rest of this post, I will detail how we use PHP, how HHVM works, the challenges we faced migrating to HHVM, and the remarkable performance wins it provides.

PHP at Box

At Box, PHP is a central part of our stack. Although we utilize a variety of languages in a large set of backend services, every request that we serve interacts with our PHP web application first. Since Box's inception, the core functionality of our product has been implemented in our PHP codebase.

Combating latency in a growing PHP codebase with more than 750,000 lines and 150 active contributors is a major challenge. We cannot cache the majority of pages we serve because users fundamentally expect mutable actions on Box to be atomic. As our product continues to grow and evolve, the natural tendency is for latency to increase. We have always invested significant effort into reducing latency, but there seemed to be no silver

bullet. We refactored old, inefficient code; pulled some self-contained components into PHP extensions; and leaned heavily upon caching shared immutable state between requests, but all made only modest incremental gains that could be easily displaced by new features. All of this changed at the start of last year when we took the time to fully evaluate HHVM.

The HipHop Virtual Machine

HHVM is an open source PHP interpreter spearheaded by Facebook. It was originally born as a PHP to C++ transpiler heavily tailored toward Facebook's PHP codebase, but in recent years it has transformed into a just-in-time (JIT) compiler. In brief, JIT compilers work by compiling blocks of PHP code that are executed frequently in a consistent manner down to assembly. Its evolution into a JIT compiler has allowed HHVM to achieve near-equivalent functionality with the normal PHP interpreter, as it now supports more dynamic aspects of the PHP language. For example, the old PHP to C++ transpiler couldn't run the "**eval**" statement in PHP ("**eval**" executes a string as new PHP code, and there is no facility in C++ to support this), whereas the new version of HHVM can. Since becoming a JIT compiler, HHVM has steadily gained popularity as a replacement for the standard PHP interpreter.

Over a year ago, we noticed that the HHVM team was putting a much larger emphasis on **achieving parity** with the normal PHP interpreter. We had evaluated HHVM in the past, but it had proved to be too hard to get HHVM to properly run our web application. This time, though, we entered into the challenge with a renewed commitment to fully evaluate HHVM and the impact it could have on latency. Incorporating HHVM into our stack and getting it to run a subset of our codebase was a significant task, but the potential rewards quickly validated our effort. Our initial experiment showed that HHVM ran a core endpoint more than four times faster than the default PHP interpreter.

This marked the start of a yearlong effort to safely migrate our production application to run on top of HHVM. During this process, we faced many different challenges in various areas of our stack. The majority of the significant obstacles we encountered were common problems others had run into as well. In the next section, I will dive into four universal obstacles on our road to HHVM: resolving unexpected incompatibilities between HHVM and the default interpreter; circumventing expected incompatibilities between the two; revamping PHP deployments; and ensuring that everything plays nicely in a mixed environment.

Achieving parity

PHP is a *huge* language. Its core runtime alone contains a massive number of functions, configuration settings, and classes that have accumulated over the course of over a decade of community contributions. This doesn't even take into account PHP's massive catalog of extensions, which all have to be ported over to HHVM. The fact that HHVM maintains near-perfect functional behavior with the default PHP interpreter is an incredibly impressive feat in itself. However, we managed to uncover a decent number of behavioral differences between the two runtimes.

We discovered the majority of runtime differences in **less popular corners** of PHP. Some of these differences were **benign missteps** that caused a unit test to fail, and some of these differences were **insidious bugs** that could have had dire consequences. HHVM achieves closer parity with the PHP interpreter every day, but migrating a big codebase is bound to surface more discrepancies in behavior. Automated testing was the single largest safety net that prevented these differences from affecting our users. Simply getting our PHPUnit test suite to pass under HHVM was a large effort that required many parity fixes. Manual testing also provided another indispensable safety net, especially to uncover bad interactions between external services and HHVM. We uncovered the vast majority of parity issues before HHVM touched production through a combination of automated and manual testing.

Fixing runtime discrepancies in HHVM is a fun process. The HHVM community is very active, and it doesn't take long to find help on **GitHub** or the **HHVM IRC Channel**. The HHVM codebase takes full advantage of modern C++ constructs and is relatively easy to understand and contribute to. Before releasing HHVM, we contributed around 20 patches for various parity issues and enhancements.

Differences by design

During the process of migration, we encountered several behavioral inconsistencies between the two runtimes that were the result of fundamental design choices. This was perhaps the trickiest class of discrepancies to work through. For example, HHVM's multiprocessing model (MPM) greatly differs from the previous MPM we used, Apache prefork. HHVM allocates one worker thread to serve a request, whereas Apache prefork allocates one worker process to serve a request. This caused a few subtle challenges. The first was a relatively simple bug to fix, once it was discovered — we have been using the

PHP process id to disambiguate logs and other temporary files. Since HHVM uses one process, the current process id no longer works to differentiate between concurrent requests. After discovering this, we did a thorough audit of all functionality in our codebase that could be affected by the new MPM, such as setting the file mode creation mask and changing directories.

A particularly unexpected behavioral difference manifested itself through the "**memory_get_usage**" function in PHP, which reports the current amount of memory allocated to a request. For certain classes of streaming requests, we use the value reported by this function to periodically flush an in-memory cache. The difference comes into play with **jemalloc**, HHVM's preferred memory allocation library. Jemalloc is a slab-based allocator, which means it allocates memory in large chunks and handles smaller allocations using those chunks. Under HHVM, "memory_get_usage" returns the total size of all slabs allocated to that request, rather than the actual amount of memory that the request is actively using in its slabs. When we continued to use "memory_get_usage" in the same way under HHVM, specific streaming requests misbehaved and thrashed their local caches, which in turn ended up driving much greater load on our backend systems. Fortunately, changing our memory reporting under HHVM to report the amount of request-local memory actively used (accomplished by setting the "$real_usage" parameter to true) completely fixed the problem.

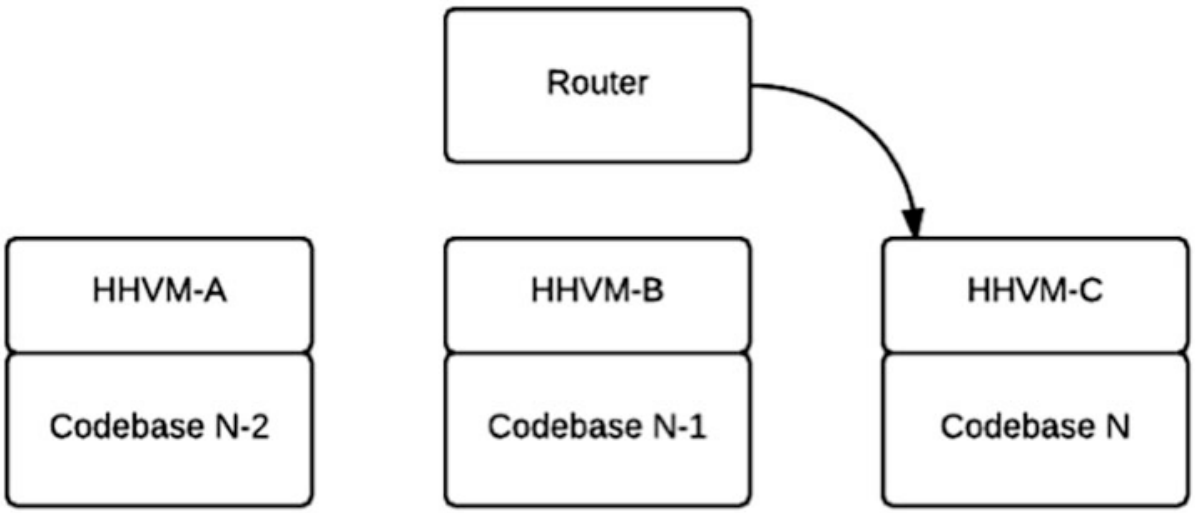
There is another, more dangerous class of issues caused by the difference in HHVM's MPM: memory leaks! In the Apache prefork MPM, slow memory leaks aren't a big concern, as worker processes can be recycled after servicing a configurable number of requests. In HHVM, this luxury no longer exists. We encountered a particularly nasty memory leak running HHVM on a nonstandard workload for multiple days. After lots of careful profiling with jemalloc, we tracked the leak down to a third-party library and promptly patched it. After this patch, HHVM remains completely stable in its memory consumption while serving millions of requests over many days on all our workloads.

Revamping deployments

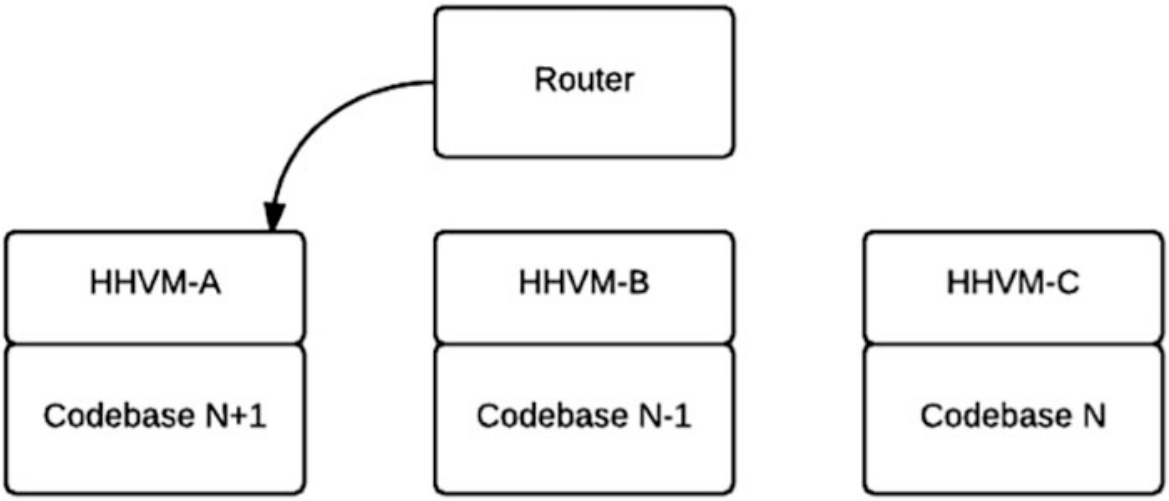
There are two fundamental differences in the operational best practices of HHVM that forced us to rethink the way we deploy our PHP application. The first difference is that HHVM needs to be "warmed up" on new code to realize its top performance. Because HHVM is a JIT compiler, it needs to run new code a few times before it can gather enough information to convert it into efficient assembly. In practice, this translates to hitting a few

important endpoints with multiple curl requests before serving live traffic. The second difference is that it's a good idea to restart HHVM on a regular cadence. This makes upgrading HHVM much easier, and it also provides a nice safeguard against memory leaks in HHVM and its linked libraries. Satisfying both of these constraints required a small paradigm shift away from the typical PHP deployment model under Apache.

We previously used one instance of the Apache webserver to serve our site, and we rotated codebases by flipping a symlink pointing to the current codebase. Now, we have three instances of HHVM ready to serve traffic at all times. In a steady state, one HHVM instance serves all traffic from the current application, and the other two are ready to serve the previous and previous-previous versions (see diagram below). Each HHVM webserver points to an absolute path, and when we want to deploy a new version, we simply stop the webserver pointing to the oldest version, start it pointing toward the new version, warm it up with multiple curl requests, and then redirect traffic to it. Using this schema, our two constraints are satisfied, and it becomes straightforward to implement rollbacks and a canary process. To rollback to a previous version, we redirect traffic toward the HHVM instance serving the previous version of code (note that this HHVM instance will already be warm). To canary traffic to a new application deployment, we simply route only a fraction of traffic to the new HHVM instance until we are confident in the new deployment. This process has proved to be robust in production handling large amounts of traffic.



An HHVM server in a steady state



An HHVM server after a code deployment

Existing in a mixed state

Not surprisingly, the most dangerous part of migrating to HHVM is rolling it out. There is no amount of testing or manual QA that can ensure HHVM will handle all requests perfectly and play nicely with downstream systems in a production environment. HHVM behaved fine under heavy testing in preproduction environments, but we remained paranoid. Furthermore, we didn't have the luxury of being able to set up a separate read-only production environment for HHVM, and we can't tolerate any amount of downtime. The only possible way for us to roll out HHVM was to do it in a controlled manner with a long, well-observed canary process. This necessitated running HHVM in the same environment as Apache and the default PHP interpreter. Being able to exist in this state ultimately enabled us to successfully release HHVM with no negative user impact.

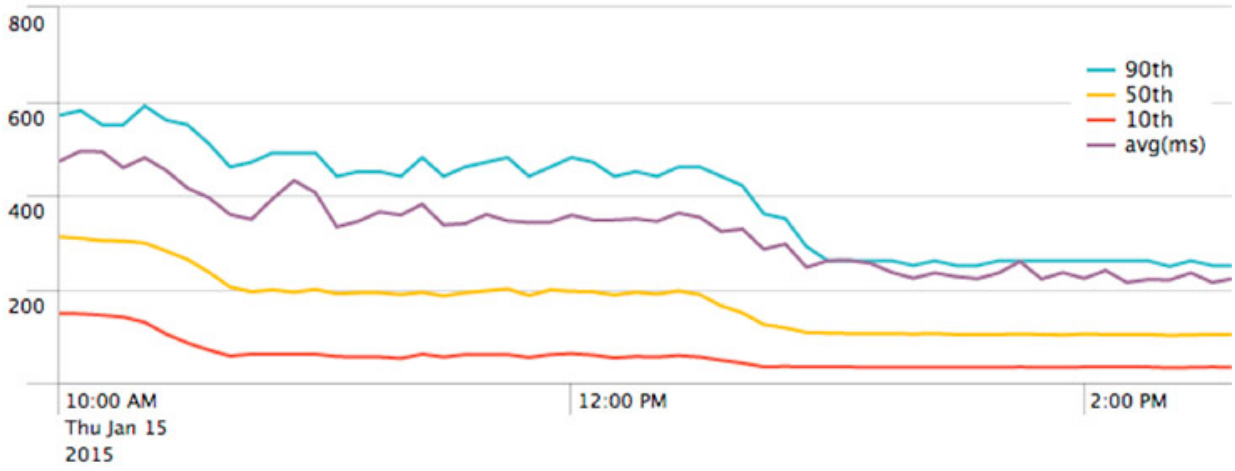
Our PHP codebase interacts with a large number of different backend systems. Fortunately, most interactions happen via curl requests to internal REST APIs, which go through a very well-tested and stable curl extension in HHVM. We use the **PDO extension** to interact with our MySQL database servers, which also exhibited parity in functional behavior with the default PHP interpreter. The potentially troublesome backend system was memcached, for the simple fact that to ensure complete interoperability between the two runtimes, both runtimes must have functionally equivalent memcached extensions and both must serialize objects identically. If either runtime serialized objects in a different manner, then it could easily wreak havoc on the other if it retrieved objects serialized in different formats from memcached. We ran numerous experiments in mixed environments to ensure that neither runtime poisoned memcached or any other backend store. Everything worked remarkably

well, except one small corner case: ArrayObjects. In the standard PHP interpreter, the extension that implements ArrayObject defines a customized serialization format, whereas HHVM just uses standard object serialization. We had to disable caching of ArrayObjects in our application to ensure memcached interoperability. Fortunately, we didn't encounter any other incompatibilities before or during the rollout.

One of the more useful facilities at our disposal during the rollout was our dead-simple host conversion methodology. Even though it's very obvious that one should make conversion to HHVM as simple as possible, it's still worth calling out at every opportunity. We decided to roll out HHVM on a host-by-host basis via puppet, and our conversion was controlled by a single flag in puppet that could be adjusted by hostname. Converting a host to HHVM and rolling back was as simple as adjusting the flag's conditional; it was not required to remove the host from a load balancer or do anything else. A complete rollback could be completed in under 5 minutes by someone half-awake at 3 in the morning. Having a very quick way to move forward and backwards on multiple hosts at once is an absolute necessity.

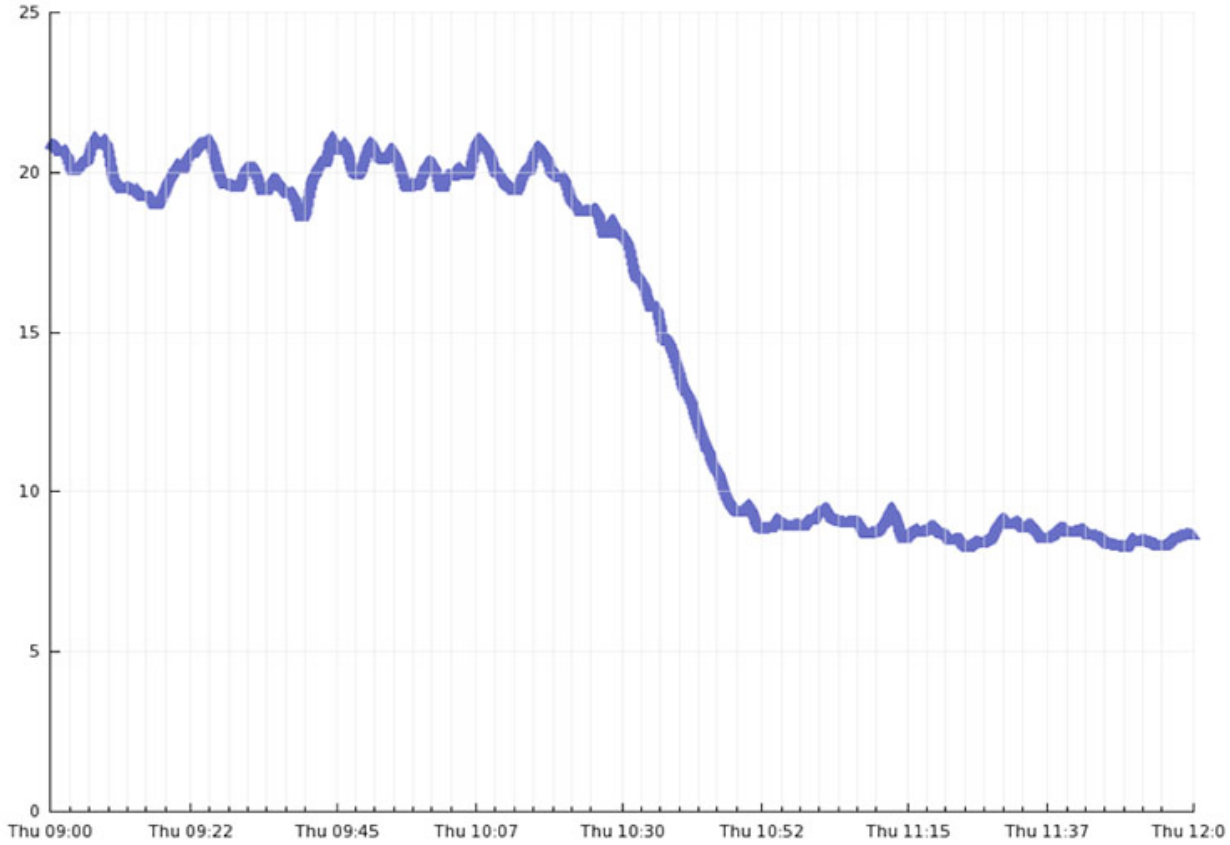
All our web application logging and monitoring systems were vital during the course of the rollout, but there was one form of monitoring in particular that proved to be especially useful: monitoring HHVM's error log for new errors. We maintain an internal database of all unique PHP errors that we have observed in Apache's error log, and we extended this system to ingest HHVM's error log as well. When rolling out HHVM, this system alerted us of any new errors that occurred under HHVM, upon which we would triage the error and use our PHP error database to judge whether this was a preexisting problem. This gave us much additional insight into whether HHVM was causing any new regressions.

The carrot at the end of the stick: HHVM's benefits



Server-side latency over the course of rolling out HHVM to the majority of our infrastructure. The rollout was approximately between 10:15am and 1:00pm.

In accordance with our early benchmarks, HHVM ultimately provided an astounding reduction in latency. The graph above shows latency over the course of the day when we rolled out HHVM to the majority of our production infrastructure. As is readily apparent, HHVM cut latency by a significant factor. On average, HHVM decreased overall server-side latency by a factor of 2.5, where server-side latency is the time spent between a request entering our infrastructure and its response leaving. This number is all the more impressive because it also includes time spent waiting on responses from backend services, and essentially all requests make many calls to many different services. There is simply no possible way we could have engineered such a dramatic drop in latency without HHVM; its gains are unprecedented. Most important, HHVM's impact on latency was very easy for the average user to notice.



Average percentage CPU utilization in one data center over the course of rolling HHVM to the majority of our infrastructure.

HHVM was also responsible for a massive gain in efficiency. Above is a graph of our average frontend CPU utilization over the course of rolling out HHVM to the majority of our servers. On average, we observed slightly over a 2x drop in CPU utilization. This gives us

over twice our current frontend capacity for free, because CPU utilization is our primary limiting factor on frontend machines. At our scale, this translates into a very significant cost savings in server expenditures, electricity usage, and reduced need for data center capacity.

HHVM also provides many other amazing features in addition to speed and efficiency, including:

- **The ability to run code written in Hack.** Hack provides many features that we desire in a language, including an expressive type system, a type checker, and support for asynchronous execution. We're seriously evaluating wholesale adoption of Hack in our PHP codebase.
- **Sophisticated profiling tools.** HHVM implements many profiling mechanisms, including those found in the XDebug extension and a time-based sampling profiler dubbed Xenon. HHVM also integrates nicely with jemalloc's memory profiling tools to provide detailed process-wide metrics and analysis.
- **More possibilities to improve speed, efficiency, and code quality.** The main facility HHVM offers in this realm is "Repo Authoritative" mode, where you can precompile PHP down to an intermediate representation and run that instead. Repo Authoritative mode forbids the use of more dynamic PHP facilities, such as evaluation of strings as PHP code. Our experiments with it have shown up to a 15 percent improvement in speed and efficiency.
- **An incredibly active community.** HHVM is making remarkably fast progress in achieving parity with the default PHP interpreter, making performance improvements, and porting over extensions. HHVM is on a quick release cadence, and the community is very responsive to issues and pull requests.

In summary, migrating PHP runtimes to HHVM wasn't an easy process, but it was well worth it. It must be done carefully, with much testing beforehand and an awareness of common pitfalls, but it's wholly possible to do it safely with zero downtime. The majority of what it takes is careful attention toward deployment architecture, conversion methodology, fixing all impactful runtime incompatibilities, and monitoring. HHVM has unbelievable potential to speed up CPU-bound PHP applications, and it also comes with many other benefits. We're incredibly excited that HHVM now powers Box, and we are still working hard toward utilizing HHVM's full potential to make Box as fast and reliable as possible.

More to Read

Sustainability @Scale 2015 recap

Recommended

Under the Hood: Building and open-sourcing fbthrift

Under the Hood: Building and open-sourcing flint

Under the Hood: warp, a fast C and C++ preprocessor

Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode

Want to work with us?

Join the team, we're hiring! Here are some of our current open positions:

Software Developer, VoIP (WhatsApp)
Software Engineer, Ads
Web Software Developer, Web Client (WhatsApp)

More Engineering Positions

Connect

Keep Updated

Stay up-to-date via RSS with the latest open source project releases from Facebook, news from our Engineering teams, and upcoming events.

Subscribe