We're always looking for ways to make our computing infrastructure more efficient, and in 2010 we deployed [HipHop for PHP](https://www.facebook.com/notes/facebook-engineering/hiphop-for-php-move-fast/280583813919) to help support the growing number of Facebook users. While HipHop has helped us make significant gains in the performance of our code, its reliance on static compilation makes optimizing our code time consuming. We were also compelled to develop a separate HipHop interpreter (HPHPi) that requires a lot of effort to maintain. So, early last year, we put together a small team to experiment with dynamic translation of PHP code into native machine code. What resulted is a new PHP execution engine based on the HipHop language runtime that we call HHVM (aka the HipHop Virtual Machine). We're excited to report that Facebook is now using HHVM as a faster replacement for HPHPi, with plans to eventually use HHVM for all PHP execution.

Facebook uses HPHPi (and now HHVM) for day-to-day software development, but uses the HipHop compiler (HPHPc) to create optimized binaries that serve the Facebook website. HPHPc is in essence a traditional static compiler that converts PHP→AST→C++→x64. We have long been keenly aware of the limitations to static analysis imposed by such a dynamic language as PHP, not to mention the risks inherent in developing software with HPHPi and deploying with HPHPc. Our experiences with HPHPc led us to start experimenting with dynamic translation to native machine code, also known as just-in-time (JIT) compilation. A dynamic translator can observe data types as the program executes, and generate type-specialized machine code. Unfortunately we didn't have a clean model of PHP language semantics built into HipHop, as HPHPc and HPHPi are based directly on two distinct abstract syntax tree (AST) implementations, rather than sharing a unified intermediate representation. Therefore we developed a high-level stack-based virtual machine specifically tailored to PHP that executes HipHop byte code (HHBC). HHVM uses HPHPc's PHP→AST implementation and extends the pipeline to PHP→AST→HHBC. We iteratively co-developed both an interpreter and a dynamic HHBC→x64 translator that seamlessly interoperate, with the primary goal of cleanly supporting translation.

Throughout the HHVM project we have tended toward simple solutions. This is nowhere more evident than in the core premise of the dynamic translator itself. Most existing systems use method-at-a-time JIT compilation (e.g., [Java](http://l.facebook.com/l.php?u=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FJava_%28programming_language%29&h=jAQEA1dDF&s=1) and [C#](http://l.facebook.com/l.php?u=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FC_Sharp_%28programming_language%29&h=ZAQHKEjgS&s=1)), though trace-based translation has also been explored in recent systems (e.g., [Tamarin](http://l.facebook.com/l.php?u=http%3A%2F%2Fwww-archive.mozilla.org%2Fprojects%2Ftamarin%2F&h=hAQF01gFz&s=1" \t "_blank) and[TraceMonkey](https://www.facebook.com/l.php?u=https%3A%2F%2Fwiki.mozilla.org%2FJavaScript%3ATraceMonkey&h=2AQG_ddIY&s=1)). We decided to try a very simple form of tracing that limits each trace to a single basic block with known input types. This "tracelet" approach simplifies the trace cache management problem, because complex application control flow cannot create a combinatorial explosion of traces. Each tracelet has a simple three-part structure:

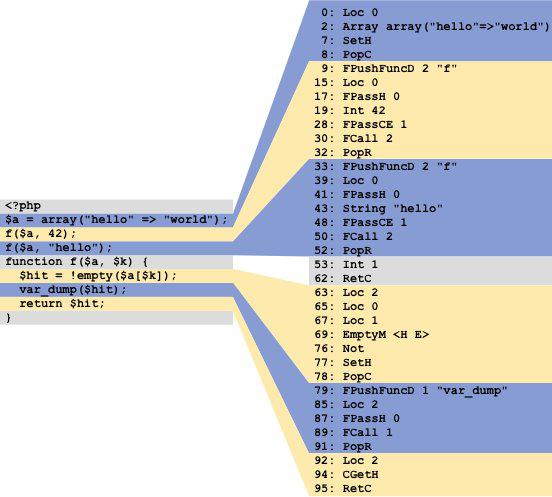
* Type guard(s)
* Body
* Linkage to subsequent tracelet(s)

The type guards prevent execution for incompatible input types, and the remainder of the tracelet does the real work. Each tracelet has great freedom, the only requirement being that it restore the virtual machine to a consistent state any time execution escapes the tracelet and its helpers. The obvious disadvantage is that tracelet guards may repeat unnecessary work. Thus far we have not found this to be a problem, but we do have some solutions in mind should a need arise.

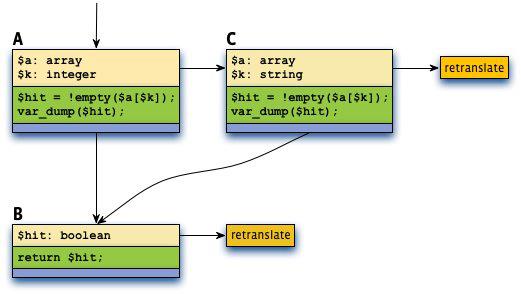
For those who are interested in understanding HHBC in detail, the [bytecode specification](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Fhhvm%2Fblob%2Fmaster%2Fhphp%2Fdoc%2Fbytecode.specification&h=bAQFEMilV&s=1" \t "_blank) is available in the HipHop source tree. However, detailed knowledge should not be necessary for understanding the following example:

f() is executed twice, for which the translator creates three tracelets total as shown below. f($a, 42) causes creation of tracelets A and B, and f($a, "hello") causes creation of tracelet C; B is used by both invocations.

How well does HHVM work? As compared to HPHPi, the HHVM bytecode interpreter is approximately 1.6X faster for a set of real-world Facebook-specific benchmarks. Right now there is a stability gap between the HHVM interpreter and translator, which precludes us reporting translator performance for the same set of benchmarks. However, we can infer from a set of benchmarks based on the [Language Shootout](http://l.facebook.com/l.php?u=http%3A%2F%2Fbenchmarksgame.alioth.debian.org%2F&h=wAQHGj4GK&s=1) that translator performance is closer to HPHPc-compiled program performance than to interpreter performance, as indicated by the geometric mean of the benchmarks (rightmost column in the following figure). The interpreters are all roughly 0.2X as fast as HPHPc, and the translator is approximately 0.6X as fast. For perspective on why this matters, consider that many Facebook engineers spend their days developing PHP code in an endless edit-reload-debug cycle.



Example PHP program translated to HHBC

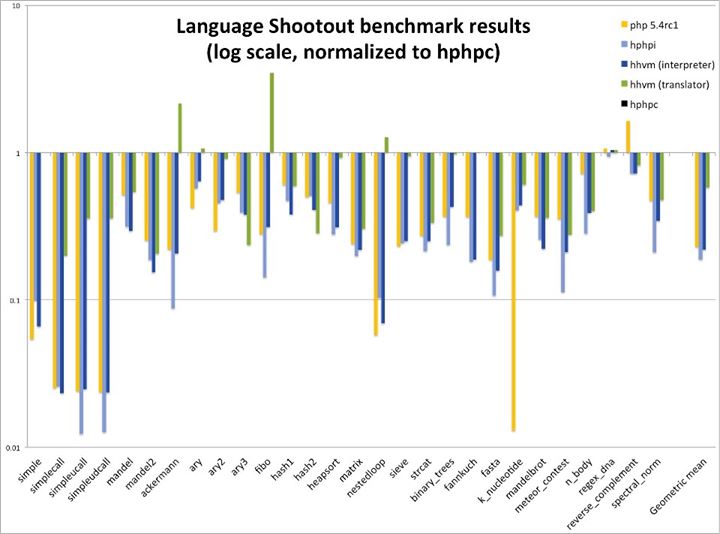


Tracelets induced by f($a, 42) and f($a, "hello")

The difference between 8-second and 5-second reloads due to switching from HPHPi to the HHVM interpreter makes a big difference to productivity, and this improvement will be even more dramatic once we enable the translator.

We expect HHVM to rapidly close the performance gap with HPHPc-compiled binaries over the coming months as the dynamic translator stabilizes and matures. In fact, we predict that HHVM will eventually outperform statically compiled binaries in Facebook's production environment, in part because we are already sharing enough infrastructure with the static compiler that we will soon be able to leverage static analysis results during tracelet creation.

Many challenges remain, as well as some uncertainty regarding the translator's behavior when running the entirety of Facebook's PHP codebase. In the near term we need to stabilize the translator and create an on-disk bytecode format (to reduce startup time and to store global static analysis results). Then we will need to optimize/tune both the translator and the interpreter as we observe how the system behaves under production workloads. Here are just a couple of the interesting problems we will soon face:



* The x64 machine code that the translator generates consumes approximately ten times as much memory as the corresponding HHBC. CPU instruction cache misses are a limiting factor for the large PHP applications that Facebook runs, so a hybrid between interpretation and translation may outperform pure translation.
* The translator currently makes no use of profile feedback, though we do have sample-based profiling infrastructure in place. Profile-guided optimization is an especially interesting problem for us because Facebook dynamically reconfigures its website between code pushes. Hot code paths mutate over time, and portions of the x64 translation cache effectively become garbage. One solution may be to repeatedly create a new translation cache based on the previous one, taking advantage of recent profile data. This approach has a lot in common with semi-space garbage collection.

The first 90% of the HHVM project is done; now we're on to the second 90% as we make it really shine. The HHVM code is deeply integrated with the HipHop source code, and we will continue to share HipHop via the public [GitHub site for HipHop](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Fhhvm%2Fwiki%2F&h=EAQExWTM0&s=1" \t "_blank), just as we have since HipHop's initial release in early 2010. We hope that the PHP community will find HHVM useful as it matures and engage with us to broaden its usefulness through technical discussions, bug reports, and code contributions. We actively monitor our [github site](https://www.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Ffacebook%2Fhhvm%2Fwiki%2F&h=FAQEJjTig&s=1" \t "_blank).

*Jason Evans is a software engineer on the HipHop team at Facebook, one of nearly 20 people who have contributed to the HHVM project so far.*