The HipHop Virtual Machine

Keith Adams Jason Evans Bertrand Maher Guilherme Ottoni Andrew Paroski
Brett Simmers Edwin Smith Owen Yamauchi

Facebook

{kma,je,bertrand,ottoni,andrewparoski,bsimmers,smith,oyamauchi}@fb.com

Abstract

The HipHop Virtual Machine (HHVM) is a JIT compiler and runtime for PHP. While PHP values are dynamically typed, real programs often have *latent types* that are useful for optimization once discovered. Some types can be proven through static analysis, but limitations in the ahead-of-time approach leave some types to be discovered at run time. And even though many values have latent types, PHP programs can also contain polymorphic variables and expressions, which must be handled without catastrophic slowdown.

HHVM discovers latent types by structuring its JIT around the concept of a *tracelet*. A tracelet is approximately a basic block specialized for a particular set of run-time types for its input values. Tracelets allow HHVM to exactly and efficiently learn the types observed by the program, while using a simple compiler. This paper shows that this approach enables HHVM to achieve high levels of performance, without sacrificing compatibility or interactivity.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – compilers, incremental compilers, run-time environments, code generation, optimization

General Terms Design, Languages, Performance

Keywords PHP; dynamic languages; JIT compiler; tracelet

1. Introduction

PHP is a popular language for server-side web application development [33]. As a late-bound, dynamically typed language, PHP is difficult to run efficiently.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2585-1/14/10.
http://dx.doi.org/10.1145/2660193.2660199

The most popular PHP implementation is a straightforward interpreter [25]. While interpreters have the benefits of simplicity and portability, they incur high CPU overheads. As shown in Section 5, these overheads limit the performance of some of the most important and common PHP applications.

In this paper, we present the HipHop Virtual Machine (*HHVM*),[1] a JIT and runtime for PHP designed with the following goals:

1. **Compatibility**. Run large and complex PHP programs, whose development history was unrelated to that of HHVM, without modification.
2. **High performance**. Efficiently map PHP programs to the hardware, weakening PHP's flexible primitives to cheaper ones where compile-time and run-time analyses allow.
3. **Predictable performance**. Provide programmers with a navigable performance landscape. Local changes should have local effects, which are unsurprising in direction and magnitude.
4. **Developer productivity**. Support the interactive development workflow that PHP developers expect, without human-visible compile or link phases for integrating changes.

The major contributions of this paper are:

- A novel solution to the type discovery problem faced by all dynamic language JITs, in the form of *tracelets*.
- Quantitative evidence of HHVM's effectiveness at improving performance of industrial, popular open source, and synthetic workloads.
- Quantitative evaluation of the monomorphicity of a multi-million line, real-world PHP application.
- Enumeration of the challenges PHP presents to efficient implementation, and a possible set of solutions.
- Overall presentation of the design and implementation of an industrial-strength, server-side dynamic-language virtual machine.

[1] HHVM is available as open source from [11].

2. Motivation for JIT Compilation

The HipHop compiler (*HPHPc*) [38] had greatly improved the performance of PHP applications before work began on HHVM. HPHPc uses ahead-of-time techniques to translate PHP into C++ and then produce a native binary. While HPHPc's performance gains over interpreted implementations are impressive, the ahead-of-time model introduces both performance limitations and operational complexity which motivated the development of HHVM.

2.1 Performance Limitations

Much of HPHPc's power to accelerate PHP comes from type inference, which allows static binding of many dynamic invocation sites. The undecidability of type inference limits the power of this technique in an ahead-of-time setting. Since the directions of many program branches are undecidable, unknown types frequently taint the data-flow graph, preventing optimization. As a just-in-time compiler, HHVM has access to the runtime values flowing through the program. In particular, the tracelet abstraction, described in detail in Section 4.2, enables HHVM to exploit type regularities that cannot be proven statically.

2.2 Operational Issues

Ahead-of-time compilation forces developers to compile and link after every program change. Since HPHPc relies on sophisticated global analysis, its compile-link cycle consumes tens of minutes for multi-million line applications, even after considerable work on compile performance. This is a poor fit for the rapid prototyping style to which PHP is suited.

To avoid these long compile cycles, HPHPc users develop in a simple interpreter called *HPHPi*. HPHPi is built alongside HPHPc, and shares as much of its runtime as possible. However, HPHPi and HPHPc are very different execution engines, and they often execute PHP differently. This gives rise to a difficult class of application-level bugs, which hide under the interpreter, and appear only when compiled. Since compilation is not part of the every-day workflow of developers, these bugs are often discovered only at deployment time, sometimes after code has already been released.

A JIT compiler like HHVM can serve both developers' desires for rapid iteration and the production environment's needs for high performance, while avoiding the correctness risks of developing and deploying in different environments.

3. PHP Challenges

PHP presents some difficulties for HHVM's goals by its very nature, as discussed below.

3.1 Features

PHP is a large language. In addition to its original procedural features, it has acquired classes, interfaces, traits, exceptions, generators, and closures. Any language implementation covering all these features is a large engineering effort.

3.2 High Performance

PHP has evolved under pressures that favor developer productivity over machine efficiency. Thus, the language resists efficient implementation. Two of the more prominent obstacles to performance are dynamic types and late binding.

Dynamic types. All user-visible PHP values share the same union type, which spans integers, booleans, floating-point numbers, strings, PHP's idiosyncratic "array" aggregate type, and user-defined classes.

Late binding. Every web request in PHP begins with an empty heap and global namespace aside from a small number of system-defined variables, functions, and classes. The mapping of names to functions and classes is populated dynamically, by unrestricted, Turing-complete PHP code.

3.3 Predictable Performance

When optimizing towards benchmarks, it is tempting to introduce a patchwork of targeted optimizations to the compiler. If these optimizations are too narrow, the application programmer is left with a peaky performance landscape, with few high peaks and many broad, low valleys. Realizing the best performance on such a VM can require unnatural coding, and intimate understanding of JIT internals. HHVM accepts, wherever possible, a lower performance ceiling in return for a higher performance floor. This goal particularly drives the trade-offs relating to tracelets in Section 4.2.2.

HHVM also strives to keep the development environment's performance as close to production's as possible. This property makes gross, naked-eye performance feedback a part of developers' regular edit/run/debug cycle.

3.4 Rapid Development

During development, it is common for developers to edit code between server requests and expect to exercise the new code immediately. When code is changed between requests, some old code is invalidated, and new code must be generated. To maintain interactive performance, HHVM avoids human-observable compilation overheads where possible.

4. The HipHop Virtual Machine

HHVM implements a virtual instruction set called HipHop bytecode (HHBC). HHVM's front-end parses and analyzes PHP source code, compiling into HHBC. HHVM's JIT compiler, interpreter, and runtime then execute HHBC code. In general, the JIT compiler is used rather than the interpreter for most execution. Figure 1 depicts the high-level design.

PHP is most commonly used in server-side web applications. We distinguish two important PHP use cases. In the *production* use case, the same (unmodified) PHP source files are executed repeatedly to serve web requests originating from real end users. To perform well in production environments, HHVM employs some of the same ahead-of-time techniques that the HPHPc compiler pioneered [38].

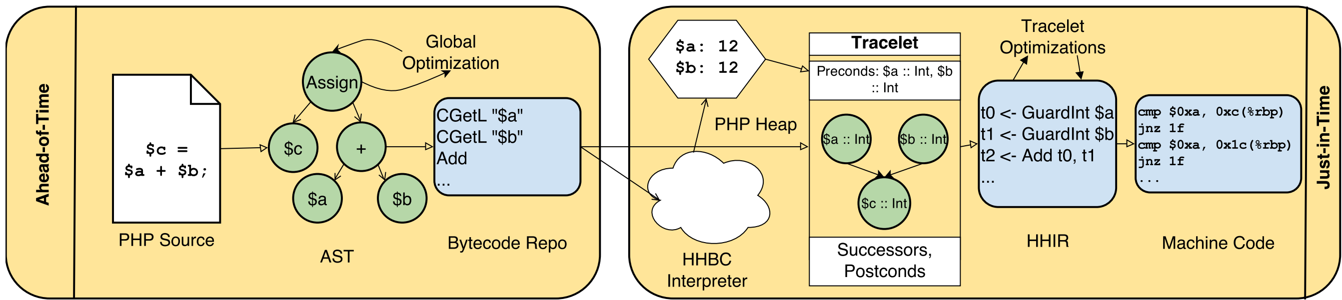

Figure 1. HHVM's production compilation pipeline.

In the *development* use case, a developer runs his/her private copy of the PHP source files (commonly referred to as a *sandbox*). During development, PHP source files can change in between requests as the developer iterates, tests, and tunes their code. To preserve PHP's developer workflow in this scenario, HHVM was built to accommodate incremental recompilation as source files change. This disallows some of the global optimizations performed in the production scenario, while still providing good responsiveness between requests by avoiding recompilation of the entire application.

4.1 HipHop Bytecode

HipHop bytecode (HHBC) defines an instruction set and a metadata schema for serializing PHP programs into a form that is easy for interpreters and just-in-time compilers to consume. Each HHBC instruction is encoded using one or more bytes, where the first byte is the opcode and any subsequent bytes encode immediate arguments. HHBC's metadata encode function and class definitions from a source file using the `Func` and `PreClass` structures, respectively. The metadata also include tables encoding identifiers and string literals, mappings to source files and line numbers, and other data needed by HHVM during execution.

HHVM is a stack-based virtual machine. HHBC instructions push and pop temporary values on the evaluation stack. It is a HHBC invariant that, at each instruction boundary in the bytecode, the depth of the evaluation stack is the same for all possible control-flow paths. Local variables are referred to using integer IDs and are assumed to be live for the duration of the function.

HHBC serves as a decoupling boundary between the frontend and backend of HHVM. The frontend can transform PHP into HHBC without executing it, and the backend can execute HHBC without access to the original PHP source files. This separation keeps HHVM's backend isolated from various frontend details, including PHP's evaluation order, control-flow semantics, and exception handling.

4.1.1 Late Binding

Reflecting PHP's untyped, late-bound nature, HHBC is generally untyped and uses names to refer to functions, classes, constants, and properties. HHBC supports converting a single PHP source file into bytecode and metadata in isolation without the need to access the definitions or contents from other source files. As a consequence, HHBC's `PreClass` structure only captures the contents of the class that are directly declared in the class statement from the source file. `PreClasses` record the names of referenced classes, traits, and interfaces, but not the contents of their definitions.

4.1.2 Program Values

PHP's unique reference feature ("&") has far-reaching effects on PHP values and parameter passing, described in detail in [34]. HHBC's instruction set encodes where references may and may not appear on the evaluation stack, which allows the JIT to generate more efficient code in the common case where references are impossible. HHBC enforces this by annotating each instruction's stack outputs and inputs with *flavors*. HHBC invariants ensure that, at each point in the bytecode, the evaluation-stack depth and the flavor of each live evaluation-stack slot are statically known and consistent with the expected flavors of the instructions consuming them.

4.1.3 Calling Convention

As an emergent consequence of references and late binding of function calls, HHVM's calling convention is complex. As shown in Figure 2, it is not statically knowable which positional arguments are passed by reference and which are passed by value. Because the by-reference or by-value nature of each positional parameter is denoted in the callee and not at the call site, and because the callee is not generally statically known at a call site, PHP call sites must identify the function to be called before evaluating parameters.

To capture these semantics, HHBC splits the calling process into three phases:

1. **Identify the callee**. HHVM resolves the callee function by whatever means are needed. HHVM records the identity of the function for which arguments are being evaluated in an activation record on the evaluation stack.
2. **Evaluate positional arguments**. With the identity of the function established, HHVM can determine which

```
if (date('l') == 'Friday') {
  function weirdArg(&$x) { $x = 'surprise!'; }
} else {
  function weirdArg($x)  { return $x + 1; }
}
$init = 13;
weirdArg($init); // Ref on Fridays; value otherwise
```

Figure 2. Value/reference semantics for positional arguments are undecidable.

```
$a = goldbachs_conjecture() ? 3.14159 : "a string";
```

Figure 3. PHP type inference is undecidable.

arguments are to be passed by reference and which are to be passed by value.

3. **Transfer control**. Finally, control can be transferred to the destination function.

4.2 JIT Compiler

HHVM's JIT compiler is responsible for discovering the source program's latent types, and using these types to generate efficient machine code.

4.2.1 Type Discovery

Zhao et al. [38] identified *type inference* as a key enabler of PHP optimization. If static analysis of the program can bound the set of types for a program value, the value can be represented untagged, or in a register, and operations on the value can be statically bound. In dynamic languages like PHP, offline type inference is generally undecidable. Consider the PHP program fragment in Figure 3. In order to resolve `$a`'s type, the program must actually be run. But Goldbach's Conjecture is either true or false, so for all real executions `$a` is either always a floating-point number or always a string.

While Figure 3 may seem contrived, similar situations arise from data pulled from databases (where the type is stable because the database schema does not change often), or from builtin runtime primitives. For example, the PHP builtin `strlen` returns an integer when passed a string, but it returns the special value `null` when passed an array. Correctly functioning applications, however, only pass strings to `strlen`, so they only see integers returned from it. In all these situations, the program has extremely predictable types that cannot be resolved via static analysis. As demonstrated in Section 5.4, programs normally have *latent types*, even when those types cannot be statically inferred.

4.2.2 Tracelets

HHVM unifies its handling of control-flow and type discovery by compiling at an unusual granularity: rather than compiling whole files, methods, or traces as most dynamic-language JITs do, it operates on *tracelets*. A tracelet is a single-entry, multiple-exit region of the source program, annotated with the types of all input values that flow into the region. The JIT symbolically executes the HHBC instructions comprising the tracelet, performing a single-pass, forward data-flow analysis. This symbolic execution annotates each HHBC instruction with input types and output types. Tracelet formation is summarized in Algorithm 1.

Algorithm 1 Tracelet construction. $vstate$ contains a partial symbolic model of program state, mapping storage locations to types.

repeat
  $preState \leftarrow vstate$
  for all $i \in inputs(pc)$ **do**
    if not $i \in vstate$ **then**
      $vstate_i \leftarrow$ type of i
      $guards_i \leftarrow$ type of i
    end if
    if $vstate_i$ is indeterminate **then**
      break
    end if
  end for
  for all $out \in outputs(pc)$ **do**
    $vstate_{out} \leftarrow$ inferType($pc, out, vstate$)
  end for
  $postState \leftarrow vstate$
  $traceletInstrs$.append(
    TraceletInstr($pc, preState, postState$))
  $pc \leftarrow advance(pc)$
until pc is control flow
return Tracelet($traceletInstrs, guards, vstate$)

Any input types that were not derived internally from the symbolic execution are resolved by observing the program's run-time state at JIT-compile time. The JIT then emits code that uses this observed state as a prediction for the types that the program will see going forward. In other words, HHVM uses the run-time types observed at JIT time as predictors for all future run-time types. As shown in Section 5.4, this crude heuristic is wildly successful, because programs tend to be overwhelmingly monomorphic. This heuristic also obviates expensive value-profiling approaches that build up a high-resolution statistical model of program types [6, 26].

When generating machine code for a tracelet, the JIT must first *guard* to ensure that the preconditions that drove the symbolic execution of the tracelet hold at run time. If the guards pass, the JIT compiler is free to optimize the tracelet body with all the type information harvested during symbolic execution, as well as to perform classic compiler optimization (constant propagation, dead code elimination, etc.) within the tracelet body. Since Algorithm 1 constructs each `TraceletInstr` with sufficiently precise types for its inputs, the JIT never needs to emit completely generic machine code; guards instead direct execution to a type-specialized version.

```
function max2($a, $b) {
   return $a > $b ? $a : $b;
}
echo max2(2, 1) . "\n";
echo max2("wxy", "abc") . "\n";
```

Figure 4. Example illustrating control-flow and polymorphism.

```
     .function("max2")
a:   CGetL $b
     CGetL $a
     Gt
     JmpZ c
b:   CGetL $a
     RetC
c:   CGetL $b
     RetC
```

Figure 5. HHBC for `max2`. Tracelet entry points have been marked with labels a, b, and c.

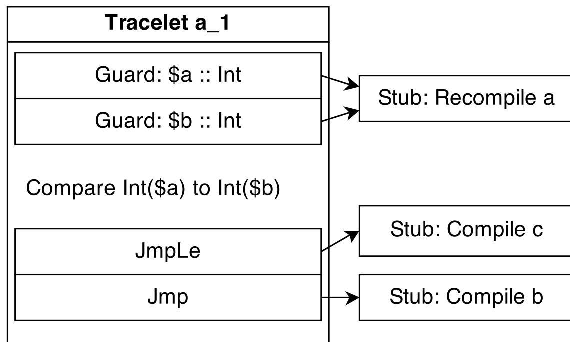

Figure 6. Contents of code cache after compiling the first tracelet of Figure 5.

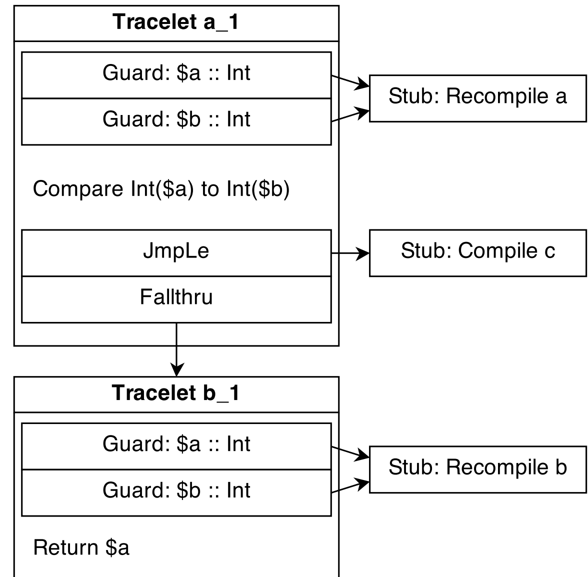

Figure 7. Contents of code cache after execution of `max2(2, 1)`.

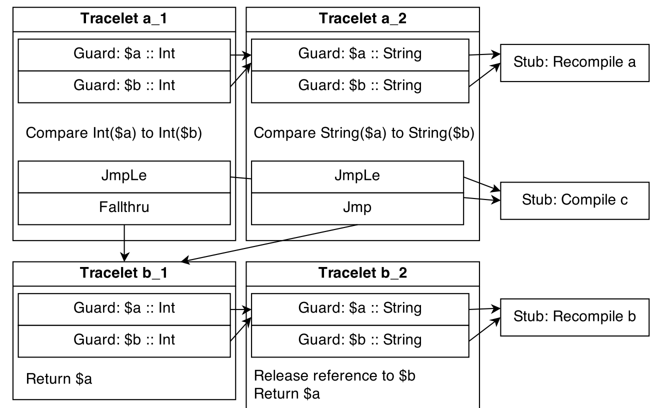

Figure 8. Contents of code cache after execution of `max2("wxy", "abc")`.

4.2.3 Leaving Tracelets

At tracelet end, HHVM uses the well-known technique of *chaining* to transfer control to the tracelet's successor(s). In the worst case, this involves a simple branch instruction. In the common case where a tracelet's successor is compiled immediately after the first run of the tracelet, execution flows through the bottom of the predecessor into the successor.

When a tracelet guard fails for the first time, HHVM branches to a recovery routine that simply reruns Algorithm 1, forming a new tracelet for the new run time types observed. The guard failure branches are then patched to point to the new tracelet translation. So, for modestly polymorphic code, the chain of tracelet guards performs a linear search for a machine code fragment that matches the current input types. Nearly all searches terminate after considering one or two candidates, as shown in Section 5.4. To limit the possible explosion of such machine fragments for tracelets with many, highly polymorphic inputs, we empirically determined to limit the length of such tracelet chains at 12. In the very rare case that the search fails after 12 candidates, HHVM resorts to the bytecode interpreter.

4.2.4 Tracelet Example

As a concrete example of the tracelet mechanism, consider the PHP fragment in Figure 4. It defines a `max2` function that computes the greatest of two input values using PHP's polymorphic > operator. When applied to integers or floating-point values, > has the usual numeric meaning, but when applied to strings it performs a lexical comparison. HHVM's front-end converts the body of `max2` to the HHBC code shown in Figure 5.

The first execution of max2 passes two integers. Algorithm 1 creates a tracelet that begins at the function's head and continues until the first control-flow instruction. The tracelet guards on the types of its two inputs, so that in the body it can use the hardware's integer comparison instruction to drive the branch. Guard failure chains to helpers that recompile this tracelet. Since there are not yet any compilations for either successor of the tracelet, both sides of the branch temporarily point to helpers that will compile the respective successors. This leaves the code cache in the state shown in Figure 6.

In the example execution of `max2(2, 1)`, the `JmpLe` is not taken, so HHVM compiles the tracelet starting at b as depicted in Figure 7. The return instruction in b ends tracelet formation after just two instructions. The tracelet again guards on the types of both local variables since the compiler needs to know if any reference-counting operations are required when leaving the function's scope. The predecessor's branch is fixed up to point to the new compilation of b; since the two were compiled back-to-back, no branch is necessary, and execution flows from a_1 directly into b_1. At completion of the first call to `max2`, the code cache looks like Figure 7.

When the program invokes `max2` on strings, the first guard in a_1 fails, causing a to be recompiled. Algorithm 1 runs again, producing a_2, specialized to operate on strings. a_2 then jumps to b_1, where the first guard fails, causing a recompilation of b. b_2 is created, which releases references to the strings in `$a` and `$b` and then returns the value in `$b`. This leaves the code cache in the state shown in Figure 8.

Any further invocations of `max2` on integers or strings with the first value greater than the second will execute rapidly now, without any further compilation. As new types are encountered, a's tracelet chain will grow until it captures the types the program naturally uses, or until it hits the configurable limit for tracelet chain length.

4.2.5 Tracelet Summary

Tracelets offer the following benefits:

1. **Complete type specialization**. Machine code only ever operates on fully disambiguated PHP values; integers are added directly in registers with the hardware's `add` instruction.
2. **Compact machine code in presence of polymorphism**. While polymorphism is unusual, it does occur. Trace trees [13] can also achieve complete type specialization, but they can also lead to code explosions [3]. A control flow graph like $A_1 \rightarrow A_2 \rightarrow \ldots A_n$ that experiences two different morphs would lead to 2^n machine code translations under trace trees, while it produces $2n$ translations in the tracelet approach.
3. **Graceful degradation in presence of polymorphism**. When the JIT's type guesses are violated at run time, HHVM does not discard entire method bodies and recompile from scratch. Instead, it slows down the polymorphic site by a single extra guard.
4. **Low, predictable compile overheads**. The JIT compiler is operating on a very small scope, usually smaller than a source basic block, so super-linear algorithms still complete quickly.
5. **Code locality**. Dead code is eliminated as an emergent property, because it is never reached. We pack compiler output into the shared cache of compilations sequentially. This has many favorable consequences. Code that is run early is also run often, so the first few pages of the compilation cache are much hotter than the last pages. This provides a kind of emergent hot/cold splitting. Since the JIT interleaves the bodies of methods, the body of a caller often "wraps" the body of methods that it calls in a natural way, which optimizes the layout of single-caller methods.

In return, tracelets incur the following costs:

1. **Limited compiler scope**. General compiler optimizations are only applicable intra-tracelet. Even loop induction variables are written back to memory at back edges and re-read at loop heads, for example.
2. **Frequent, redundant guards**. Every tracelet checks the types of its inputs. If a local variable is frequently accessed, and its type could not be inferred ahead-of-time, it may get checked once for each tracelet entered. While some limited inter-tracelet analysis can, in theory, exploit post-conditions of the predecessor to skip guards in the successor, we have not been able to show a performance win in real PHP programs from this technique and have avoided it for sake of simplicity.
3. **Inability to use common compiler infrastructure**. LLVM [19] has democratized high-quality compiler infrastructure to a great extent, but it is inherently a method-at-a-time system. At present, LLVM is not geared towards compiling "open" method bodies of the sort required by HHVM's tracelet approach.

4.2.6 Tracelet JIT Internals

The HHVM JIT compiles tracelets into machine code by representing the tracelet's HHBC instructions and type information in a lower-level intermediate representation, called *HHIR*. HHIR is based on the static single assignment (SSA) form [10], and was specifically designed for HHVM. HHIR's key design decisions include:

1. **Typed:** Given the JIT philosophy of only performing computation on specialized types, HHIR is a typed representation (unlike HHBC).
2. **PHP-aware:** HHIR enables not only traditional compiler optimizations, but also important PHP-specific optimizations.

3. **Portable:** HHIR is machine-independent, which reduces the complexity of targeting different architectures. The JIT currently supports x86-64, and an ARMv8 backend is under development.
4. **Low-level:** Despite being machine-independent, HHIR is close enough to typical microprocessors to enable efficient machine code generation directly from HHIR. This allows HHVM to produce efficient code in the absence of target-specific representations and optimizations.

Once a tracelet is translated to HHIR, a number of classic compiler optimizations are performed, including constant propagation and folding, common-subexpression elimination, dead code elimination, and jump optimizations.

In addition to these classic optimizations, HHIR also supports several unusual optimizations. A tracelet-specific optimization is *guard relaxation*, which attempts to unspecialize tracelet guards for values that only feed instructions that do not benefit from more precise type information. Guard relaxation increases the rate of guard success, reduces the number of guards needed, and reduces the length of tracelet chains.

Another important optimization that we identified for PHP is *reference-counting elimination*. PHP uses reference-counting [9] to manage memory. Unfortunately, naive reference counting is costly, and there are several ways in which a program can observe PHP's use of reference counting [34]. To cope with this, HHIR explicitly represents reference count manipulations, and eliminates them when an optimization pass proves them to be unnecessary.

After HHIR optimization, the compiler performs register allocation via the extended linear scan algorithm [35], and finally generates machine-code.

4.3 Runtime System

Solving PHP's unique challenges requires careful attention in the runtime library, affecting HHVM's loading and execution of functions, classes, constants, and other PHP constructs.

4.3.1 The Empty World Programming Model

PHP's programming model starts every web request in an "empty world", with a namespace containing only a smattering of special global variables and the standard library. It is the PHP code's responsibility, on each request, to bind names to the user-defined classes, functions, and constants that it wishes to use. In a multi-million line codebase, with many tens of thousands of classes and functions, this mechanical process of reconstructing a mostly-identical global namespace imposes a significant overhead on each request.

This empty-world model means that PHP's binding of names to program objects is both dynamic (not determinable at compile time) and fluid (liable to change from request to request). Developers may define multiple classes or functions with the same name, as long as at most one of each name is loaded in a single request. As shown in Figure 2, the developer may bind different definitions to the same name along different control flow paths. HHVM fully supports this dynamicity, while aggressively optimizing performance for the common case.

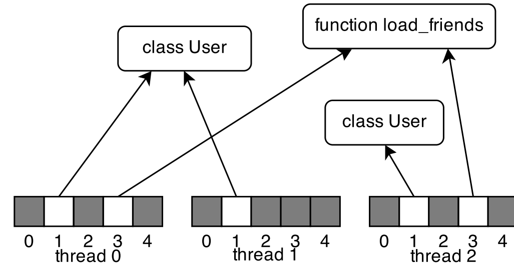

Figure 9. Illustration of the different definitions of the same name available in different threads. The definition of the class `User` is at offset 1 in every thread's target cache, while the definition of the function `load_friends` is at offset 3. Threads 0 and 1 share the same definition of User and thread 2 references a different `User` class. Threads 0 and 2 share the same definition of the `load_friends` function, while thread 1 has no definition.

4.3.2 Request-Local Lookup: the Target Cache

The definition of a class can vary from request to request, so without some kind of global analysis, HHVM needs an efficient per-request map from names to entities (functions, classes, and constants). HHVM's solution is the *target cache*, a request-local section of memory with a globally defined layout. The target cache holds information that must be quickly accessible but may differ between requests, as illustrated in Figure 9. HHVM allocates fixed offsets in the target cache to represent a given named entity. In this example, offset 1 has been chosen to refer to the `User` class, and offset 3 refers to the `load_friends` function. Thread 1 does not have an entry at offset 3, which means that `load_friends` has not yet been defined in thread 1's request. This shared layout allows HHVM to emit code that can trivially look up the definition of a given class or function, regardless of how or when it was defined.

4.3.3 `PreClass`es and `Class` Reification

As mentioned in Section 4.1, HHVM's `PreClass` structure encodes the contents of a class statement from the source file. `PreClasses` record the names of referenced classes, traits, and interfaces, but not the contents of the corresponding definitions, which may possibly be defined in other source files.

When a class definition from a source file is loaded at run time, HHVM accesses the `PreClass` and looks up all the parent classes, interfaces, and traits transitively and builds a `Class` structure that represents the full class with all its inherited methods, properties, constants, etc. HHVM's backend primarily uses `Classes`, and `PreClasses` are typi-

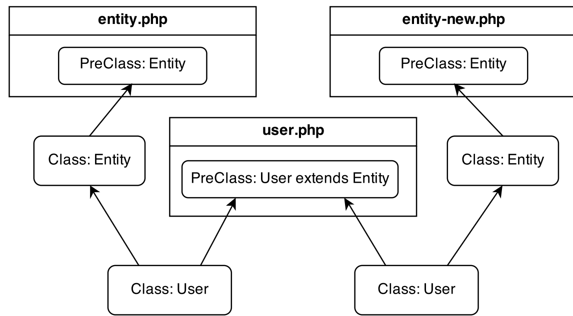

Figure 10. Example illustrating the relationship between PreClasses and Classes. The files entity.php and entity-new.php each define their own version of the Entity class, and the file user.php defines a User class that derives from Entity. This results in two distinct definitions of a class named User from a single PreClass.

cally only used when loading PHP source files and reifying Classes.

This split minimizes the amount of recompilation needed when changing a single file. Consider a scenario where two different definitions of a class Entity exist, in files named entity.php and entity-new.php. A third file, user.php, contains a class User, which extends entity. Figure 10 shows one potential outcome after a few different requests have executed. Notice that there are two different instantiations of the User class, each referencing one of the two Entity classes. Both User classes, however, reference the same PreClass.

If the developer now changes entity.php and reloads a page referencing that file, HHVM would only have to recompile entity.php. The User class is reified by combining the User PreClass, created completely from user.php, with its parent, the Entity Class. This ability to incrementally recompile a subclass of a popular parent class, without recompiling all files depending on the parent class, helps greatly with interactive performance in sandbox mode.

4.3.4 Fast Property Access via Func Cloning

Another challenge caused by PHP's malleable class namespace is that a single textual definition of a PHP method may have different behavior between requests. For example, consider the PHP fragment of Figure 11.

The getName() method simply returns the object's name field. However, because of the dynamic definition of User's parent class, the name field may reside at different offsets from the beginning of the object in different requests. Performing a dynamic lookup of name on every run time access to the field would be quite expensive relative to a simple offset-load.

To resolve this problem, HHVM uses Func cloning. As mentioned in Section 4.1, HHVM uses the Func structure for representing a PHP function or method. *Cloning* a Func

```
if (f()) { class Named { var $name; var $a; } }
else     { class Named { var $a; var $name; } }

class User extends Named {
  function getName() {
    return $this->name;
  }
}
```

Figure 11. Accessing a property at an undecidable offset.

means creating a copy that shares most, but not all, of its attributes with the original. Every Func created in the system has a unique, 32-bit ID. When reifying a PreClass into a Class, the runtime ensures that each combination of base class and subclass has a unique cloned Func for each of its methods. The ID of each of these Funcs may be combined with a bytecode offset to form a SrcKey, which uniquely identifies a program location.

By ensuring that a given SrcKey uniquely identifies both a method and certain attributes of the class containing that method, HHVM is able to generate significantly more efficient code. Returning to the example of the name property, this allows HHVM to assume that the property referred to by the expression $this->name will always live at the same fixed offset from the beginning of the object. If the User class is ever instantiated with a different parent class in a different request, the new Func for User::getName() will have a different ID, making it a distinct source location from the JIT's perspective.

4.3.5 Ahead-of-Time Analysis

The target cache and Func cloning are critical for good performance in sandbox mode, but in production mode HHVM can do even better. Production mode begins with an ahead-of-time global analysis. All the PHP in the codebase is parsed and loaded into memory. The types of local variables, object properties, function return values, and other program locations are inferred when possible, classic compiler optimizations such as dead code elimination and constant propagation are performed, and the resulting bytecode and metadata is serialized into an on-disk database called the *repo*. In production mode, the repo is treated as the only PHP code that exists. If there is only one class in the repo named UserProfileLoader, HHVM can assume that any reference to a class named UserProfileLoader will either use the class from the repo or throw a fatal error because the class has not yet been loaded. If there is only one function in the repo named check_friend_count and it always returns an integer, HHVM can assume that any call to a function names check_friend_count will either throw an exception or return an integer.

The ability to use these assumptions at run time allows HHVM to emit more efficient code. The most valuable of these optimizations is the ability to statically bind method

calls. However, there is a penalty to flexibility: safely depending on these assumptions requires banning constructs such as `eval()` in production mode. While a very limited form of `eval()` is theoretically possible in production mode, allowing user code to arbitrarily create new classes and functions at run time could violate invariants discovered during ahead-of-time analysis.

4.3.6 Autoloading

Production mode allows HHVM to make powerful assumptions, but it still must handle the "empty world" starting state of each request. If HHVM assumes that uniquely-defined functions and classes exist from the beginning of the request, it could visibly change program behavior. However, the performance gained by avoiding function and class loading on every request is too significant to ignore.

HHVM makes it possible for PHP programmers to avoid these overheads by extending PHP's *autoloading* functionality. PHP already allows registration of an autoloader callback in user code, which the runtime calls when a non-existent class is referenced. The autoloader function is given the name of the missing class and is responsible for loading the file defining that class. Once the autoloader returns, the operation requiring the class continues. HHVM supports the standard autoloader, and optionally extends the facility with support for undefined functions and constants as well.

HHVM also uses information from ahead-of-time analysis to tell the runtime which classes and functions are *unique* and *persistent*. We call a class or function *unique* if it does not share its name with any other class or function, and *persistent* if it is defined unconditionally at the top level of a file, and the transitive closure of classes, interfaces, and traits it depends on are also persistent.

Armed with information about which entities are unique and persistent, the runtime can construct a list of classes and functions which are to be treated as though they are defined from the beginning of time. This slightly modifies the "empty world" assumption, so HHVM users must opt into this behavior. Facebook's production experience has been that this greatly reduces the fixed startup costs associated with each request.

5. Evaluation

We assess HHVM's performance by comparing it to `php`, the popular interpreter-based implementation that serves as a *de facto* language standard [25], and to HPHPc, the PHP-to-C++ translator previously developed and used at Facebook [38]. We report the performance of each engine in terms of its speedup over `php`. To avoid confusing the language with one of its implementations, we write "PHP" when referring to the language, independent of its execution engine, and "`php`" to refer to the standard, interpreted implementation. Unless stated otherwise, all experiments are performed with HHVM 2.4.1, `php` 5.5.9, and the last version of HPHPc that can be built from the HipHop source tree. The operating system is the 64-bit version of Ubuntu 12.04 server. The system is a 4-vCPU VMware Fusion [1] 6.0.4 virtual machine, running on an 8-core Intel(R) Core(TM) i7-4960HQ CPU @ 2.60GHz.

5.1 Microbenchmarks

To measure the performance of each PHP engine in isolation, we use a suite of 37 microbenchmarks. This suite includes the popular Programming Language Shootout benchmarks [32] and a hand-constructed collection of benchmarks called `vm-perf`, which is designed to exercise aspects of whole-program behavior that are not stressed by typical small benchmarks. For example, since HHVM seeks good performance on large PHP programs, `vm-perf/big` is an artificial program that is too large to fit in instruction cache. Similarly, `vm-perf/cache_get_scb` is a 3,000-line kernel extracted from an old version of Facebook's data-fetching layer, which once accounted for a significant fraction of Facebook CPU utilization.

The results for `php`, HPHPc, and HHVM in both its sandbox (HHVM) and production (HHVMr) configurations are summarized in Figure 12. HHVM achieves a geometric mean speedup of $2.64\times$ in sandbox mode and $2.79\times$ in production mode, compared to HPHPc's speedup of $1.93\times$. On a few microbenchmarks, `php` outperforms HHVM due to superior implementations of a few runtime library functions: `regex_dna` is dominated by regular expression matching (`preg_match` and `preg_replace`), and `revcomp` is dominated by `wordwrap`, which HHVM implements in pure PHP and `php` implements natively. The main loops of `dyncall` and `mandelbrot_1`[2] are written in the top level scope outside of any function bodies, which HHVM does not JIT due to aliasing issues unique to that scope. HHVM outperforms `php` on benchmarks that depend mostly on the PHP execution engine, such as `binarytrees` and `spectralnorm`.

5.2 Open-Source Applications

As a test of both language compatibility and performance, we evaluate three popular PHP applications. These experiments use the nginx web server [22], configured to use either `php` or HHVM via FastCGI. For each application, we followed the recommended installation procedure, generated some content (several wiki pages for Mediawiki, several photos and documents posts for Wordpress and Drupal), and benchmarked the end-to-end latency visible from a client running on the host system via `ab` [31].

The results of this experiment are summarized in Figure 13. In sandbox mode HHVM attains a $1.6\times$ speedup over `php`. HPHPc is absent from these results because it cannot run these applications, and HHVM cannot run Drupal correctly in production mode.

[2] The `mandelbrot_1a` benchmark was mechanically derived from `mandelbrot_1` by moving its body into a function. All three new engines run `mandelbrot_1a` much faster than `php`.

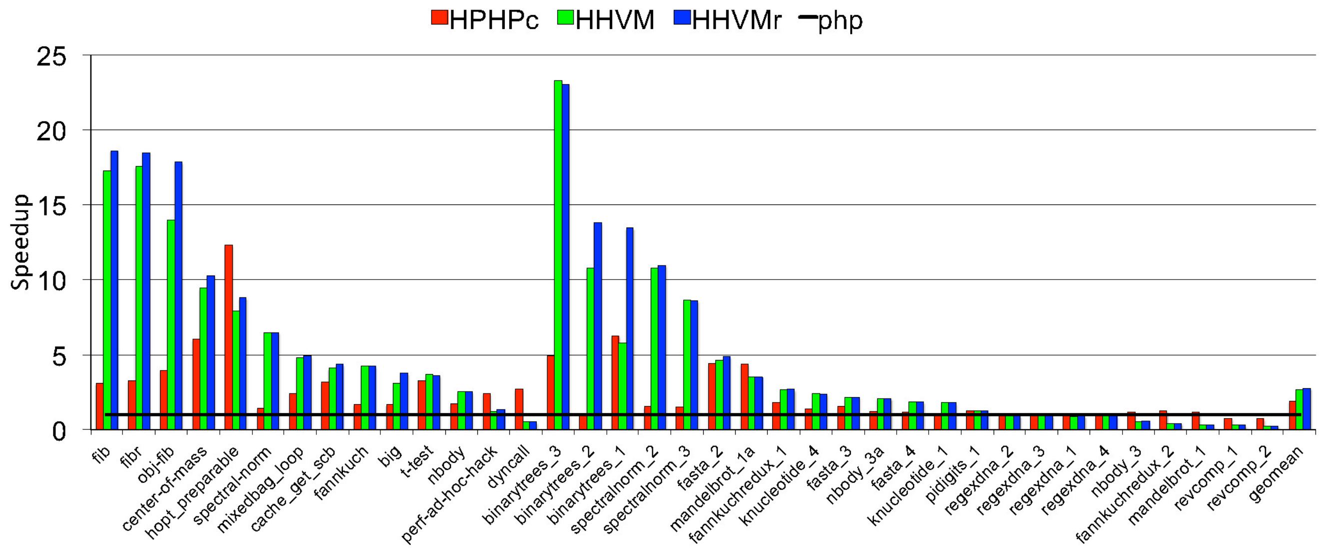

Figure 12. Microbenchmark performance as speedup over php, whose performance is normalized to 1.0.

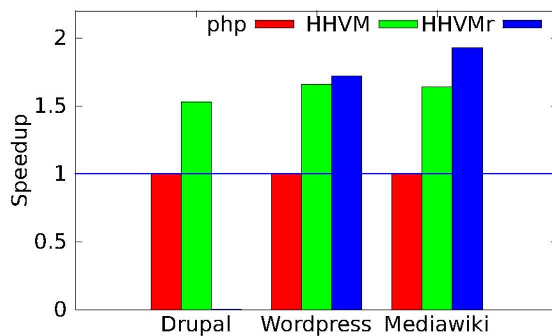

Figure 13. Performance comparison on open-source applications. HHVMr is unable to run Drupal correctly.

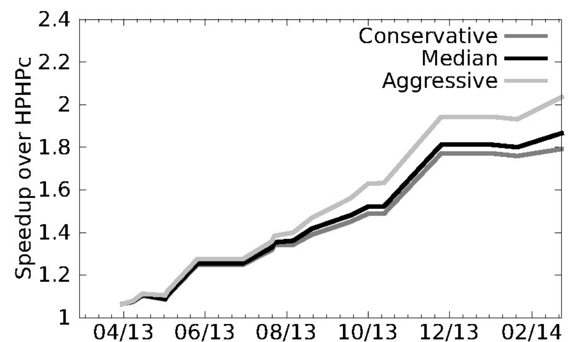

Figure 14. Efficiency gains over time for running production `facebook.com` since the initial release of HHVM. The three curves reflect the results of picking the most optimistic, median, and most pessimistic of three experiments at each release.

5.3 Facebook Production

Facebook develops a multi-million LOC PHP codebase used by more than 1 billion people worldwide, which we will refer to as `facebook.com`.

Unfortunately, a clear engine-to-engine comparison on `facebook.com` is not easy. php has not been able to run `facebook.com` since 2010, when HPHPc became operational [38]. For a few months in early 2013 both HHVM and HPHPc were able to run `facebook.com`, and thus we can compare the two engines directly during this time period. At this point, HPHPc consistently required 10-12% more CPU time to handle a given load, so the baseline comparison conservatively assumes that HHVM is 10% more CPU-efficient than HPHPc.

In March 2013, `facebook.com` began to use language features that were not available in HPHPc, so direct comparisons using newer versions of `facebook.com` are impossible past this point. Operational issues make it impossible to run older versions of `facebook.com`, so we cannot directly compare the current version of HHVM with HPHPc on an older version of `facebook.com`. We instead compare newer versions of HHVM with older versions of HHVM, and reason from those comparisons back to the versions of HHVM that were directly comparable to HPHPc.

We measure the efficiency of new releases of HHVM using an internal system called Perflab. Perflab replays real web requests from production in a controlled environment to isolate the performance effect of changes in `facebook.com` or HHVM. When preparing each new release of HHVM, we run three Perflab runs comparing the new and old releases. The compounded results of these comparisons are plotted over time in Figure 14. Taking the most pessimistic of each

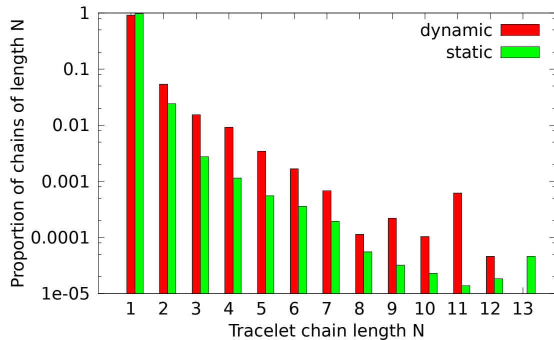

Figure 15. Distribution of tracelet-chain lengths. Note the log scale. Static refers to the entire population of compiled tracelet chains, and dynamic counts the number of steps in tracelet searches at run time. 99.5% of static tracelet chains have two or fewer members. 99.3% of tracelet walks conclude after considering 4 or fewer tracelets.

of the three experiments, HHVM has increased its efficiency $1.8\times$ since it was directly comparable to HPHPc. Compounding this with its 10% initial advantage over HPHPc, we estimate that HHVM is twice as CPU efficient as HPHPc. Using a similar methodology, Zhao et al. estimated in [38] that HPHPc was approximately $5.8\times$ more efficient than the significantly older version of php that HPHPc replaced. While this indirect comparison is inherently problematic, it suggests that HHVM runs `facebook.com` $11.6\times$ more efficiently than the version of php that HPHPc replaced.

5.4 Tracelet Polymorphism

Tracelet compilation is most effective when most code has a small number of latent types. In a tracelet-based JIT, polymorphism manifests as long tracelet chains. To test the hypothesis that short chains are predominant, we instrumented a build of HHVM and examined the occurrence of long tracelet chains running `facebook.com`.[3] We summarize the results of this study in Figure 15.

Examining the static count of tracelet chains, short chains dominate: 97% of chains have length one, and 99% have length one or two. This data confirms the hypothesis that almost all code receives a narrow set of types.

Examining the dynamic occurrence of tracelet chains, we observed that polymorphic code is more frequently executed, so the length of tracelet chains searched at run time is, on average, longer than the length of tracelet chains emitted by the compiler. However, the distribution of chain lengths is still dominated by short chains. 91% of tracelet chain searches succeed on their first try, and 99% probe 4 or fewer tracelets. Approximately one in every 21,000 tracelets executed hits the maximum tracelet chain length of 12, and so is executed by the interpreter.

[3] This experiment was performed with a current version of HHVM, since 2.4.1 is unable to run `facebook.com`.

6. Related Work

HHVM builds on a rich tradition of JIT-compiled dynamic programming language implementations. Smalltalk [15] and SELF [17] pioneered the area, and we have directly applied variants of inline caching to method dispatch in HHVM. Those investigating the SELF [17] dynamic language also observed and exploited the stable latent types flowing through naturally occurring dynamic programs.

Most modern dynamic-language JITs use a compilation strategy that can be grouped into one of two families: either method-based, or trace-based. Method-based dynamic-language JITs include Smalltalk, SELF, JavaScriptCore [2], and V8 [16]. Two prominent examples of trace-based JITs are LuaJIT [24] and TraceMonkey [13].

6.1 Dynamic Binary Translators

The tracelet abstraction was partially inspired by basic-block-at-a-time dynamic binary translation (*DBT*) systems. DBT is often useful in systems closer to the hardware/software interface, where the source program is not available. Some of the DBT systems that use basic-block-at-a-time techniques include VMware's virtual machine monitor [1], the Embra machine simulator [37], the Shade instruction-level profiler [8], and the DynamoRIO dynamic compilation framework [14].

These DBT systems must solve the code/data disambiguation problem, of differentiating instruction bytes which are executed by the CPU from those that are simply read and written. Since machine code is free to interleave code and data arbitrarily, the only completely reliable signal that a sequence of bytes is meant as instructions rather than data is when the CPU's instruction pointer refers to them. So DBT systems often compile a single-entry, single-exit unit of source machine code at a time.

The VMware VMM also uses its basic-block-at-a-time compilation technique to amortize the costs of handling self-modifying code. When it detects that source code is frequently modified, it amends the machine translation by prepending guards to each native basic block that check that the current source code matches the source code observed at compile time. These guards are analogous to the tracelet guards in HHVM. HHVM's treatment of *polymorphic* code is very close to machine-level DBTs' treatment of *self-modifying* code. While a DBT's basic-block-at-a-time strategy limits compilation scope, and the guards for self-modifying code impose overheads, these costs are acceptable in DBTs, which inspired the idea that they might be acceptable in a dynamic language context as well. However, although we have not explored in this paper, it is possible that PHP execution will benefit from larger compilation units as some DBTs have [23].

6.2 Other PHP Engines

A number of research and industrial efforts have attempted to accelerate PHP.

Ahead-of-time PHP compilers. HHVM was built directly on the source and runtime of HPHPc [38], which powered Facebook's PHP servers from 2010 to its replacement by HHVM in early 2013. HPHPc used global ahead-of-time analysis to generate optimized C++ as an intermediate representation, allowing it to leverage highly optimizing C++ compilers. The `phc` project [5] and Roadsend [29] are conceptually similar approaches that parse and analyze PHP source, and then translate it to C or C++.

While HHVM is a JIT compiler, its production mode exploits many of the same ahead-of-time optimizations present in its predecessor system, HPHPc, and these related systems.

"Repurposed JIT" compilers. Many other PHP implementers have observed that PHP's dynamicity makes offline generation of efficient machine code hard. Since JIT compilers and runtimes are notoriously complex engineering artifacts, many projects seek to leverage an existing managed-code environment such as the Java Virtual Machine (*JVM*) [20] or the Common Language Runtime (*CLR*) [21].

Phalanger [4] compiles PHP source files ahead-of-time into MSIL, the bytecode of the CLR. Quercus [27] and P9 [30] similarly compile PHP source into JVM bytecodes ahead-of-time. This approach to compilation is similar to ahead-of-time compilation to C or C++, since both the CLR and JVM are statically typed virtual machines. However, since these environments defer machine code generation to run time, the hope is that the indirections required in a static runtime will be inlined and optimized away.

This approach has the great virtue of saving considerable engineering effort. However, the semantics of PHP, and statistical behavior of PHP programs, differ enough from those of the languages for which the JVM and CLR are designed to expect that "the repurposed JIT phenomenon" identified by Castanos et al. in [7] will bound the performance of this approach relative to full-custom efforts like HHVM.

Partial evaluation approaches. A promising recent development is the emergence of dynamic language JITs based on partial evaluation. In systems like PyPy [28] and Truffle [36], dynamic-language authors write specially annotated interpreters in a host language (for PyPy, a restricted Python dialect, and for Truffle, Java). Then, a unified runtime consumes the interpreter annotations to partially inline pieces of the interpreter into a JIT compilation for the dynamic-language program the interpreter is running. This has the promise of leveraging JIT engineering effort across many languages, while avoiding the pitfalls outlined in [7].

HappyJIT [18] and HippyVM [12] are PHP implementations that leverage PyPy's partial evaluation-based JIT compiler to accelerate PHP.

7. Conclusion

HHVM is overall the best performing PHP implementation we are aware of, while being among the most complete, capable of running large open-source PHP applications, and the Facebook server-side application at significant speedup. It achieves this performance without compromising the developer workflow that makes PHP a productive environment: from the point of view of a typical web developer, its behavior is indistinguishable from that of the more familiar `php`.

HHVM is imperfect. It is still a living software system that is undergoing rapid improvement. However, its basic design of a stack-based bytecode compiled into type-specialized, guarded tracelets has been unchanged over its four-year history. While more sophisticated approaches undoubtedly have a higher performance ceiling, the tracelet JIT, despite its simplicity, provides large speedups for diverse, real PHP applications. We have found that the tracelet approach works well because natural dynamic programs use a narrow range of types per source location.

Acknowledgments

The HHVM project has been a huge engineering effort, which would not have been possible without the contributions and/or support of: Ali-Reza Adl-Tabatabai, Andrei Alexandrescu, Paul Bissonnette, Sean Cannella, Jordan DeLong, Fred Emmott, Qi Gao, Sara Golemon, Andrei Homescu, Eugene Letuchy, Scott MacVicar, Mike Magruder, Alexander Malyshev, Joel Marcey, Aravind Menon, David Mortenson, Jan Oravec, Michael Paleczny, Jay Parikh, Joel Pobar, Iain Proctor, Xin Qi, Jeff Rothschild, Dario Russi, Mike Schroepfer, Jason Sobel, Paul Tarjan, Herman Venter, Max Wang, Mark Williams, Jingyue Wu, Minghui Yang, and Haiping Zhao. Special thanks to Haiping Zhao and Mark Williams, who have made fundamental technical contributions to both HHVM and HipHop. We also thank Erik Meijer and Ole Agesen for their feedback on earlier drafts of this paper.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, October 2006.

[2] Apple. JavaScriptCore. Web site: http://trac.webkit.org/wiki/JavaScriptCore.

[3] M. Bebenita. *Trace-Based Compilation and Optimization in Meta-Circular Virtual Execution Environments*. PhD thesis, UC Irvine, 2012.

[4] J. Benda, T. Matousek, and L. Prosek. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. In *Proceedings on the 4th International Conference on .NET Technologies*, pages 11–20, 2006.

[5] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1916–1923, 2009.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 259–269, December 1997.

[7] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2012.

[8] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In T. Conte and C. Gimarc, editors, *Fast Simulation of Computer Architectures*, pages 5–46. Springer US, 1995.

[9] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12), December 1960.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] Facebook, Inc. The HipHop Virtual Machine. Web site: http://hhvm.com.

[12] M. Fijalkowski. HippyVM. Web site: http://hippyvm.com.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[14] T. Garnett. Dynamic optimization of ia-32 applications under dynamorio. *Master's thesis*, 2003.

[15] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, 1983.

[16] Google. The V8 JavaScript engine. Web site: http://code.google.com/p/v8.

[17] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, 1991.

[18] A. Homescu and A. Şuhan. HappyJIT: a tracing JIT compiler for PHP. In *Proceedings of the 7th Symposium on Dynamic Languages*, pages 25–36, 2011.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[20] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[21] E. Meijer, R. Wa, and J. Gough. Technical overview of the common language runtime, 2000.

[22] Nginx. Nginx. Web site: http://wiki.nginx.org.

[23] G. Ottoni, T. Hartin, C. Weaver, J. Brandt, B. Kuttanna, and H. Wang. Harmonia: a transparent, efficient, and harmonious dynamic binary translator targeting x86. In *Proc. of the 8th ACM International Conference on Computing Frontiers*, pages 26:1–26:10, May 2011.

[24] M. Pall. The LuaJIT project. Web site: http://luajit.org.

[25] PHP5. Web site: http://php.net.

[26] F. Pizlo and G. Barraclough. Value profiling for code optimization, Feb. 13 2014. US Patent App. 13/593,404.

[27] Quercus: PHP in Java. Web site: http://www.caucho.com/resin-3.0/quercus/.

[28] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 944–953, 2006.

[29] Roadsend compiler. Web site: http://www.roadsend.com.

[30] M. Tatsubori, A. Tozawa, T. Suzumura, S. Trent, and T. Onodera. Evaluation of a just-in-time compiler retrofitted for PHP. In *Proceedings of the ACM International Conference on Virtual Execution Environments*, pages 121–132, 2010.

[31] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool. Web site: http://httpd.apache.org/docs/2.2/programs/ab.html.

[32] The Computer Languages Benchmark Game. Web site: http://shootout.alioth.debian.org/.

[33] Tiobe. TIOBE programming community index. Web site: http://www.tiobe.com/tiobe_index/index.htm.

[34] A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. Copy-on-write in the PHP language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 200–212, 2009.

[35] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 170–179, 2010.

[36] C. Wimmer and T. Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, pages 13–14, 2012.

[37] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[38] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The HipHop compiler for PHP. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 575–586, October 2012.

A. Raw data

Here is the raw data for evaluation experiments.

A.1 Microbenchmarks

```
benchmark, hhvm, hhvmi, hhvmr, hphp, php
shootout/binarytrees_1.php,10.93,36.8,4.72,10.12,63.38
shootout/binarytrees_2.php,5.26,30.77,4.12,56.69,56.69
shootout/binarytrees_3.php,4.23,18.47,4.27,19.87,98.39
shootout/fannkuchredux_1.php,3.04,13.52,3.03,4.54,8.13
shootout/fannkuchredux_2.php,13.68,9.78,13.66,4.82,6.19
shootout/fasta_2.php,7.25,30.83,6.9,7.66,33.69
shootout/fasta_3.php,5.84,13.76,5.82,8.13,12.53
shootout/fasta_4.php,3.28,8.2,3.27,5.06,6.02
shootout/knucleotide_1.php,2.21,5.45,2.22,3.69,3.96
shootout/knucleotide_4.php,4.82,11.89,4.91,8.47,11.72
shootout/mandelbrot_1.php,57.88,42.83,57.9,14.79,17.06
shootout/mandelbrot_1a.php,4.84,42.84,4.83,3.89,17.07
shootout/nbody_3.php,11.28,10.81,11.27,5.37,6.41
shootout/nbody_3a.php,3.07,10.82,3.06,5.15,6.4
shootout/pidigits_1.php,10.55,10.67,10.56,10.54,13.53
shootout/regexdna_1.php,5.62,5.45,5.43,5.4,5.08
shootout/regexdna_2.php,17.36,17.25,17.3,17.13,16.94
shootout/regexdna_3.php,12.22,12.23,12.25,12.19,11.96
shootout/regexdna_4.php,4.89,4.62,4.9,4.75,4.47
shootout/revcomp_1.php,2.1,10.35,2.13,0.78,0.61
shootout/revcomp_2.php,2.13,10.33,2.14,0.81,0.6
shootout/spectralnorm_2.php,5.41,83.93,5.32,37.51,58.21
shootout/spectralnorm_3.php,2.13,26.44,2.15,12.41,18.47
vm-perf/big.php,14.49,56.04,11.87,26.56,44.9
vm-perf/cache_get_scb.php,2.94,8.26,2.78,3.79,12.08
vm-perf/center-of-mass.php,0.74,3.9,0.68,1.16,6.98
vm-perf/dyncall.php,2.4,1.29,2.4,0.46,1.24
vm-perf/fannkuch.php,1.68,12.55,1.69,4.23,7.17
vm-perf/fib.php,0.41,7.66,0.38,2.26,7.07
vm-perf/fibr.php,0.4,7.69,0.38,2.15,7.02
vm-perf/hopt_preparable.php,0.9,11.61,0.81,0.58,7.13
vm-perf/mixedbag_loop.php,1.94,9.38,1.9,3.82,9.36
vm-perf/nbody.php,5.36,22.5,5.36,7.83,13.6
vm-perf/obj-fib.php,0.37,5.24,0.29,1.3,5.17
vm-perf/perf-ad-hoc-hack.php,0.61,0.97,0.55,0.3,0.73
vm-perf/spectral-norm.php,0.16,1.54,0.16,0.72,1.04
vm-perf/t-test.php,21.6,96.06,21.68,23.94,79.16
```

A.2 Open Source Applications

This experiment collected thousands of data points for each combination of application, endpoint, and engine. The data can be downloaded at:
`https://dl.dropboxusercontent.com/u/2184298/all.csv`,
and it has md5 checksum:
`6e326c74f284b94f9c804531f57912d3`.

A.3 Facebook Production Improvements

```
date,worst;mid;best
2013-04-02,-0.05;0.02;0.06
2013-04-09,0.89;1.17;1.61
2013-04-16,-0.73;-0.73;-0.73
2013-05-03,0.89;1.20;1.68
2013-05-05,0.0;0.0;0.0
2013-05-27,0.1;0.2;0.6
2013-06-30,2.08;2.88;3.07
2013-07-22,3.07;3.14;3.26
2013-07-24,-1.82;-0.26;0.20
2013-08-07,0.17;0.59;1.11
2013-08-21,-1.72;-2.84;0.45
2013-09-19,-0.23;0.45;0.99
2013-10-02,0.00;0.21;0.83
2013-10-27,0.0;0.0;0.0,0.0;0.0;0.0
2014-01-03,-0.02;0.26;0.54
2014-01-23,-0.19;-0.01;0.16
2014-02-24,-0.35;-2.34;-4.35
```