The HipHop Compiler for PHP

Haiping Zhao Iain Proctor Minghui Yang Xin Qi Mark Williams Qi Gao
Guilherme Ottoni [*] Andrew Paroski Scott MacVicar Jason Evans Stephen Tu [†]

Facebook, Inc.

Abstract

Scripting languages are widely used to quickly accomplish a variety of tasks because of the high productivity they enable. Among other reasons, this increased productivity results from a combination of extensive libraries, fast development cycle, dynamic typing, and polymorphism. The dynamic features of scripting languages are traditionally associated with interpreters, which is the approach used to implement most scripting languages. Although easy to implement, interpreters are generally slow, which makes scripting languages prohibitive for implementing large, CPU-intensive applications. This efficiency problem is particularly important for PHP given that it is the most commonly used language for server-side web development.

This paper presents the design, implementation, and an evaluation of the HipHop compiler for PHP. HipHop goes against the standard practice and implements a very dynamic language through *static compilation*. After describing the most challenging PHP features to support through static compilation, this paper presents HipHop's design and techniques that support almost all PHP features. We then present a thorough evaluation of HipHop running both standard benchmarks and the Facebook web site. Overall, our experiments demonstrate that HipHop is about 5.5× faster than standard, interpreted PHP engines. As a result, HipHop has reduced the number of servers needed to run Facebook and other web sites by a factor between 4 and 6, thus drastically cutting operating costs.

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

General Terms Languages, Design, Performance

Keywords PHP, dynamic languages, compilation, C++

[*] Contact author. E-mail address: ottoni@fb.com

[†] Currently affiliated with MIT.

1. Introduction

General-purpose scripting languages like Perl, Python, PHP, Ruby, and Lua have been widely used to quickly accomplish a broad variety of tasks. Although there are several reasons for the widespread adoption of these languages, the key common factor across all of them is the high programmers' productivity that they enable [14]. This high productivity is the result of a number of factors. First, these languages typically provide a rich set of libraries, which reduce the amount of code that programmers need to write, test, and debug. Second, the dynamically-typed nature of scripting languages provides flexibility and high degrees of dynamic polymorphism. Last, and most important, dynamic languages are in general *purely interpreted*, thus providing a fast cycle to modify and test source changes. By "purely interpreted" we mean that there is no compilation step necessary to run a program after writing or modifying it. This advantage is particularly relevant in the development of very large systems, which typically take substantial time to statically compile and link.

On the down side, the major issue with scripting languages in general is performance. Because of their dynamic features, these languages are typically implemented via interpreters. Experiments have shown that programs tend to take an order of magnitude more CPU time to run when implemented in interpreted languages compared to corresponding implementations in compiled languages [22].

Although performance is often not important for small scripts written to accomplish simple tasks, it does become a major problem when developing large CPU-intensive applications. This has spurred a number of efforts to try to improve the performance of scripting languages [3, 8, 10, 19, 21, 24]. Among all general-purpose scripting languages, this problem is probably most pronounced for PHP, since it is the most commonly used language for server-side web development [23]. Given the scale of users and data that Internet companies have, improving server efficiency is a key factor to reduce their operating costs incurred by running huge data centers.

This paper describes the approach that we took at Facebook to address the performance problem of PHP. Our main insight to improve performance was to implement PHP via a *static compiler*, called *HipHop*. Although static compilation requires all the code to be known in advance, this is a minor imposition for server-side applications, which are the core usage of PHP. Going against the standard practice of implementing PHP via interpreters, HipHop has proved to be a high risk, high reward project. As this paper describes, there are a number of challenges to enable static compilation of a dynamic language like PHP. Despite these challenges, HipHop supports a very large subset of PHP, and it only lacks a few, less useful language features. HipHop is a source-to-source compiler, and it transforms PHP programs into semantically equivalent C++ programs. This design enables leveraging high-quality optimizing C++ compilers to produce machine code, and we currently use GCC [9]. To obtain high performance, in addition to leveraging GCC optimizations, HipHop also implements a number of optimizations. As our experiments show, this approach has enabled HipHop to run both benchmark and real-world applications 5.5× faster on average than the standard PHP implementation [25]. Finally, although HipHop only supports PHP, we believe that many of its techniques can be used to improve the performance of other scripting languages as well.

HipHop has been released by Facebook as an open-source project [6], and it is currently deployed not only within Facebook but also in other companies. In all these cases, HipHop has reduced the number of servers needed to run web sites by a factor between 4 and 6, thus drastically cutting operating costs.

This paper describes our experience in the design and implementation of the HipHop compiler. More specifically, this paper makes the following main contributions:

1. It describes the set of techniques that enabled implementing a general-purpose, dynamic scripting language with a static compiler.

2. It describes the set of compilation techniques that enable HipHop to achieve outstanding performance for PHP.

3. It presents a thorough experimental evaluation of HipHop running both benchmarks and real-world applications from Facebook. This evaluation includes comparisons with other PHP execution engines.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to PHP, its key dynamic features, and its standard implementation. Section 3 then presents the design of the HipHop compiler and its main techniques. Following that, Section 4 presents a detailed experimental evaluation of HipHop and compares it to other PHP execution engines. Finally, Section 5 discusses related work and Section 6 concludes the paper.

```
01)   <?php
02)
03)   define("confName", "OOPSLA");
04)   define("firstYear", 1986);
05)
06)   function year($edition) {
07)     return firstYear - 1 + $edition;
08)   }
09)
10)   echo "Hello " . confName . "'" . year(27);
```

Figure 1. PHP code example

2. PHP Background

PHP is an imperative scripting language, originally conceived in 1995 for creating dynamic web pages. After its initial definition, PHP has evolved significantly. Since version 3, PHP is an object-oriented programming language and today, at version 5.4, it is a general-purpose language used beyond web development. Like Java, PHP combines interfaces and single inheritance. However, since version 5.4, PHP also supports traits [5]. In the absence of a formal specification, the language is defined by its standard implementation [25].

Although PHP syntactically resembles C++, PHP is inherently much more dynamic. Figure 1 illustrates a simple PHP example, which prints "`Hello OOPSLA'2012`". The `define`s in lines 3 and 4 are calls to a built-in function that declares a constant named after its first argument and with value given by its second argument. In line 10, `echo` prints to the output and '.' is string concatenation. Syntactically, some key differences from C++ are: (1) names of variables start with '$', and (2) variables are not declared.

Given that C++ well represents the class of static, high-performing languages, while PHP is representative of the much worse performing scripting languages[1], it is worth pointing out the main differences between the two languages. After describing these differences in Section 2.1, Section 2.2 briefly introduces the approach used in PHP's standard implementation. This brief overview provides key insights into the performance gap between PHP and C++.

2.1 Comparison between PHP and C++

Table 1 summarizes the key language features that are more general in PHP than in C++. In general, these PHP features incur significant runtime costs, and their more restricted forms in C++ enable more efficient execution. Below, we briefly discuss and illustrate each of these features. This discussion not only gives insights into the performance gap between the two languages, but it also demonstrates the biggest challenges that HipHop faces to translate PHP code into C++.

[1] The performance gap is 36.8× for measured benchmarks using Zend PHP [22].

Feature	PHP	C++
types	dynamic	static
function and class name binding	dynamic	static
dynamic name resolution	yes	no
dynamic symbol inspection	yes	no
reflection	yes	no
dynamic code evaluation (eval)	yes	no

Table 1. Key language features that are richer in PHP than in C++

2.1.1 Dynamic Typing

Being dynamically typed, PHP allows variables to hold values from different types during the execution. For example, `$x` takes both string and integer values in the following code:

```
function foo($x) {
  echo "foo: " . $x . "\n";
}
foo("Hello");  // prints: "foo: Hello"
foo(10);       // prints: "foo: 10"
```

This is fundamentally different from statically typed languages like C++, where the programmer has to declare the types of variables, which the compiler can then rely on.

2.1.2 Dynamic Name Binding

In PHP, the names of functions and classes are only bound to concrete implementations during runtime. For example, consider the following PHP code:

```
if ($cond) {
   function foo($x) { return $x + 1; }
} else {
   function foo($x) { return $x - 1; }
}
$y = foo($x);
```

If `$cond` is true, function `foo` is defined per the first declaration and `$y` gets the value of `$x + 1`. Otherwise, the second definition of `foo` is used and `$y` gets the value of `$x - 1`.

Note that, however, PHP does not allow functions and classes to be redeclared during execution.

2.1.3 Dynamic Name Resolution

In PHP, the names of variables, properties, functions, and classes may reside in variables and may be dynamically created. Consider the following example:

```
$a = 'f';
$b = 'c';
$c = 'oo';
$func = $a . $$b;
$func();
$obj = new $c;
```

This code invokes function `foo` in the fifth line. Notice the `$$` in front of `b`: `$b` first evaluates to `c`, and then `$c` evaluates to `oo`. In the last statement, this example creates an object of class `oo`.

The following example illustrates dynamic properties:

```
class C {
  public $declProp = 1;
}
$obj = new C;
$obj->dynProp = 2;
echo $obj->declProp . "\n";
echo $obj->dynProp  . "\n";
```

Although property `dynProp` is not declared in class `C`, this program is valid and prints out `1` and `2`.

2.1.4 Dynamic Symbol Inspection

In PHP, programs can make runtime queries to check whether or not functions or classes have been declared. For instance, code like the following is valid:

```
if (function_exists('foo')) {
   ...
}
if (class_exists($c)) {
   ...
}
```

The ability to make such queries is intrinsic to dynamic languages, while static languages typically assume all the program is seen before it is executed.

2.1.5 Reflection

PHP has native support for reflection, which enables querying various properties about classes, methods, functions, and more. However, C++ does not have native support for reflection. Notice that the lack of reflection is specific to C++ and not a general limitation of static languages.

2.1.6 Dynamic Code Evaluation

Like other scripting languages, PHP supports evaluating arbitrary strings as PHP code. In PHP, this is supported through calls to the `eval` built-in function. While dynamic code evaluation is simple to support in purely interpreted languages, its support in static languages would be much more complex and thus is not normally provided.

2.2 Standard PHP Implementation

In order to support PHP's dynamic features such as those described in the previous section, PHP's standard implementation (Zend) is an interpreter [25]. The choice of an interpreter is not particular to PHP, as it is also the most common approach used to implement other dynamic languages like Perl, Python, and Ruby. In this section, we give a brief overview of how the Zend interpreter works, which provides some insights into its main sources of inefficiency.

Zend is a bytecode interpreter, meaning that it uses a lower-level program representation – the Zend bytecode.

This is in contrast to AST (abstract syntax tree) walker interpreters, which interpret a program while traversing its AST. The first time a file is invoked, Zend parses it and translates it into bytecode. Zend then interprets bytecode instructions one at a time. In its highest performing setup, Zend uses a bytecode cache (e.g. APC [1]), which avoids repeatedly parsing the program and translating it to bytecode.

As with other dynamic languages, Zend discovers and loads various program components during execution as source files are included. This feature is known as *dynamic loading*. The program components, including classes, functions, variables, and constants, are kept in various lookup tables. Dynamic loading is particularly expensive for classes, which require composing class methods, properties, and constants along with those from parent classes and traits. While composing classes, Zend also needs to perform a number of semantic checks. For instance, it needs to check if method overrides are valid, and if all interface and abstract base-class methods are implemented by concrete classes.

During execution, each time the interpreter needs to access a given symbol, it uses the symbol's name to consult the lookup tables. This incurs runtime costs referred to as *dynamic lookups*. For example, consider the code in Figure 1. The execution of the `echo` statement incurs one constant lookup (`confName`) and one function lookup (`year`), in addition to one constant lookup (`firstYear`) and one variable lookup (`$edition`) in function `year`. Although some of these dynamic lookups can be eliminated through bytecode optimizations, in practice these optimizations have resulted in very small speedups (1-2%) [12].

In addition to the costs of dynamic loading and lookups, another major overhead in Zend comes from *dynamic typing*. Since variables can contain values of different types during execution, they are kept in generic, boxed values called z-values. The Zend bytecode instructions are mostly untyped and, while interpreting each bytecode instruction, Zend checks the types of operands in order to perform the appropriate actions. For example, the `ADD` bytecode first checks if the operands are numbers and, if not, attempts to convert them according to a set of type-specific rules. The mere type check itself amounts to a very significant overhead for the common case of adding two integers.

3. The HipHop Approach

Historically, there has been a rough separation between "static" and "dynamic" languages. This categorization is based mostly on whether the languages are statically or dynamically typed, but it also encompasses other languages features such as those listed in Table 1. While dynamic languages provide higher levels of abstraction and thus tend to increase productivity, static languages are commonly used when performance is important. In fact, the categorization between static and dynamic languages is usually aligned with how the language is implemented. Static languages are usually statically compiled, which significantly helps achieving high performance. In contrast, dynamic languages are generally interpreted, which helps supporting their dynamic features but significantly affects performance.

HipHop breaks this standard practice and implements PHP via static compilation. With this approach, HipHop brings high performance to a typical scripting language.

In Section 3.1, we give an overview of HipHop's approach and discuss how it addresses the main PHP inefficiencies. After that, Section 3.2 presents the high-level design of HipHop. Finally, Section 3.3 describes the techniques that HipHop uses to address the challenges imposed by PHP's dynamic nature.

3.1 Overview

Being a static compiler, HipHop is fundamentally different from PHP's standard implementation. This key difference has several implications. First, it requires all the source code to be known a priori. For server-side applications, where PHP is mostly used, we believe this is a small imposition. Also, knowing all the source code in advance enables whole-program analyses that can significantly boost performance. Second, most but not all PHP features are supported by HipHop. Most notably, HipHop does not support dynamic code evaluation (`eval`). Support for `eval` is theoretically possible in HipHop by invoking an interpreter. However, we opted not to implement that because the use of `eval` exposes serious security problems and is therefore discouraged [15, 18]. For this same reason, HipHop does not support PHP's `create_function()` and `preg_replace()` with "/e". Besides `eval`, HipHop does not support automatic promotion from integer to floating-point numbers in case of overflow. The decision not to support this PHP feature was made in the interest of performance. Third, HipHop effectively analyzes, compiles, and loads all symbols before execution starts. Although this differs from original PHP semantics, which only *hoists* (i.e. pre-loads) symbol definition under certain ill-defined rules, we have found that HipHop's semantics is easier to reason about and more often matches the programmer's intent. Finally, static compilation incurs a significant cost to rebuild large systems even after small code changes, which can dramatically reduce programmer productivity. To address that, one can combine the use of HipHop for building production code with the use of interpreters for code development. This is the setup we have been successfully using at Facebook.

Although HipHop's approach faces a number of challenges to support dynamic language features (discussed in Section 3.3), it directly addresses the three main PHP inefficiencies discussed in Section 2.2. Below, we explain how static compilation addresses each of these overheads.

Dynamic Loading. By compiling the entire program in advance, HipHop can reduce most of the overhead of dynamically loading classes, functions, and variables. This not only

Compiler Phase	Role
1. Parsing	generates AST
2. Program analysis	collects symbols and their dependencies
3. Pre-optimizer	performs type-independent optimizations
4. Type inference	infers primitive types of expressions
5. Post-optimizer	performs type-dependent optimizations
6. Code generation	outputs C++ code

Table 2. HipHop compiler phases

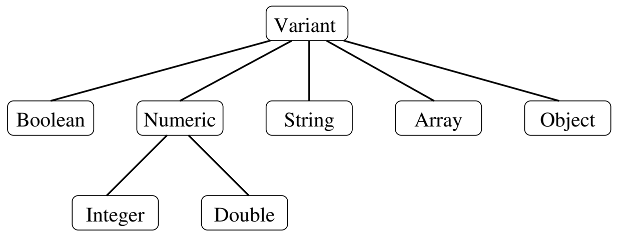


Figure 2. HipHop type hierarchy

enables classes to be composed statically, but it also enables creating various lookup tables for functions, classes, and variables during compilation instead of execution time. For the cases where PHP's dynamic-loading semantics matter, we have designed a set of techniques to provide that semantics (see Section 3.3), but which incur runtime costs. Overall, this combination of techniques allows HipHop to eliminate overheads in the common case while still supporting PHP semantics.

Dynamic Lookups. Static compilation also eliminates most of the overhead associated with dynamic lookups. In the common case, the compiler can statically resolve symbol names and emit code that directly accesses program components (e.g. variables, functions). This enables the generation of efficient binary code where the addresses of components are used, thus saving expensive dynamic table lookups. For the rare cases that the compiler cannot statically resolve symbol names, HipHop employs less efficient techniques based on lookup tables (see Section 3.3).

Dynamic Typing. As a static compiler, HipHop can perform more extensive program analyses than is practical at runtime. This enables HipHop to do aggressive *type inference*. For variables and expressions that HipHop is able to infer primitive types, this information is used to produce efficient code that bypasses dynamic type checking.

3.2 High-level Compiler Design

HipHop is a source-to-source compiler, and it uses an AST as the primary program representation throughout its phases. HipHop has six main compilation phases, listed in Table 2. Below, we briefly describe HipHop's steps at each of these phases, and illustrate its operation in Figure 3 for the example from Figure 1.

Phase 1 – Parsing
The parser reads in the PHP source files and creates corresponding ASTs, which are then used by subsequent phases. Figure 3(c) illustrates the AST constructed for the PHP code in Figure 3(a).

Phase 2 – Program Analysis
This phase traverses the ASTs and collects information about all symbols, including classes, functions, variables, constants and more. With this information, HipHop can determine how many symbols are declared with the same name, which then enables it to generate more efficient code for uniquely defined symbols. The information collected for program-defined constants is shown at the lower left corner of Figure 3(c). Besides collecting this information, this phase also creates a dependence graph between program symbols. This dependence graph is used by later phases for various purposes and also enables their parallelization.

Phase 3 – Pre-optimizer
This phase performs a number of optimizations that do not require type information. These optimizations include: constant inlining and folding; logical-expression simplification; dead-code elimination; copy propagation; inlining of small user-defined functions, and also built-in functions like `class_exists()` and `defined()` that can be statically determined; string concatenation optimizations. Given that PHP scripts generally manipulate a lot of strings, string concatenation is very common and worth special attention. PHP's double-quoted strings may contain variables that need to be replaced. The pre-optimizer breaks these strings into cascaded calls to concat. In addition, naïve pairwise concatenation is inefficient because it requires memory allocation and data copying at each step. To mitigate that, the pre-optimizer replaces cascaded pairwise concatenations with a call to an internal `concat()` function that performs a series of concatenations at once.

Figure 3(d) shows the AST after it is processed by the pre-optimizer. In function `year`, the value of constant `firstYear` is replaced and the result of the subtraction is folded. In the `echo` statement, the call to function `year` is inlined, which allows the addition `1985 + 27` to be folded. Finally, all the operands in the resulting chain of string concatenations become known constants, so the resulting string is concatenated at this phase.

Phase 4 – Type Inference
This phase attempts to determine the types of various symbols, and it plays a central role in HipHop. The more types HipHop can statically infer, the fewer checks need to be performed at runtime. HipHop uses an adaptation of the traditional Damas-Milner constraint-based algorithm [4] to infer the types of constants, variables, function parameters and return values, and other expressions. The HipHop built-in types are illustrated in the type hierarchy in Figure 2. *Variant* is the most general type, to which all symbols belong before type inference. For symbols whose types can be in-

```
<?php

define("confName", "OOPSLA");
define("firstYear", 1986);

function year($edition) {
  return firstYear - 1 + $edition;
}

echo "Hello " . confName . "'" . year(27);
```

(a) PHP Code (hello-oopsla.php)

```
const StaticString k_confName("OOPSLA");
const int64 k_firstYear = 1986LL;

/* SRC: hello-oopsla.php line 6 */
Numeric f_year(const Variant& v_edition) {
  return (1985LL + v_edition);
}
Variant pm_php$hello_oopsla_php(bool incOnce,
                                LVariableTable* variables,
                                Globals *globals) {
  echo("Hello OOPSLA'2012");
  return true;
}
```

(b) C++ Code

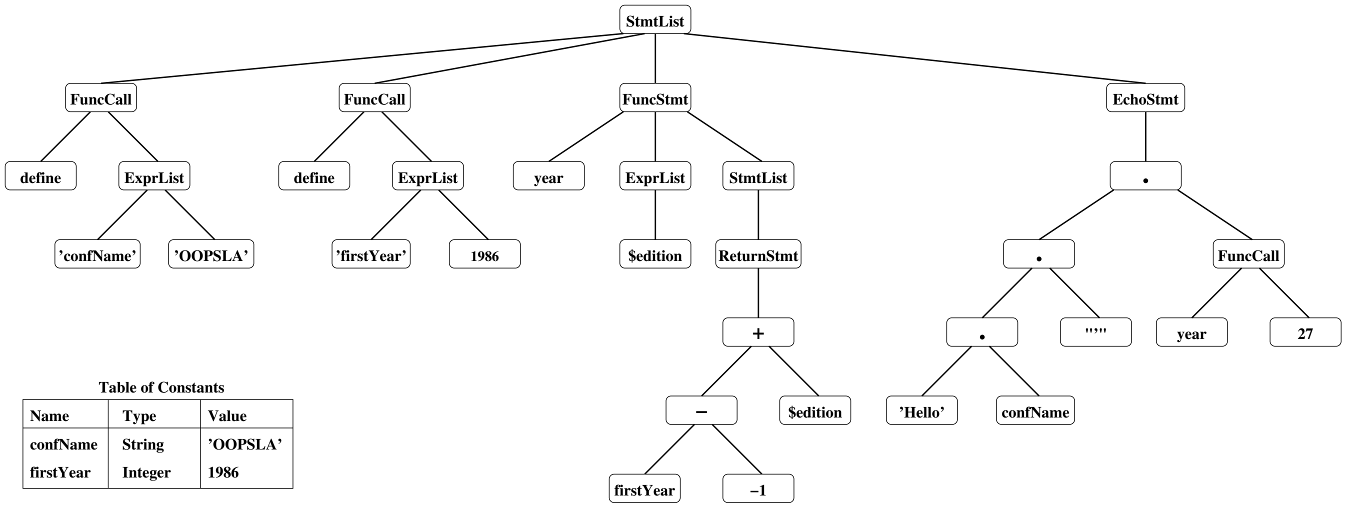
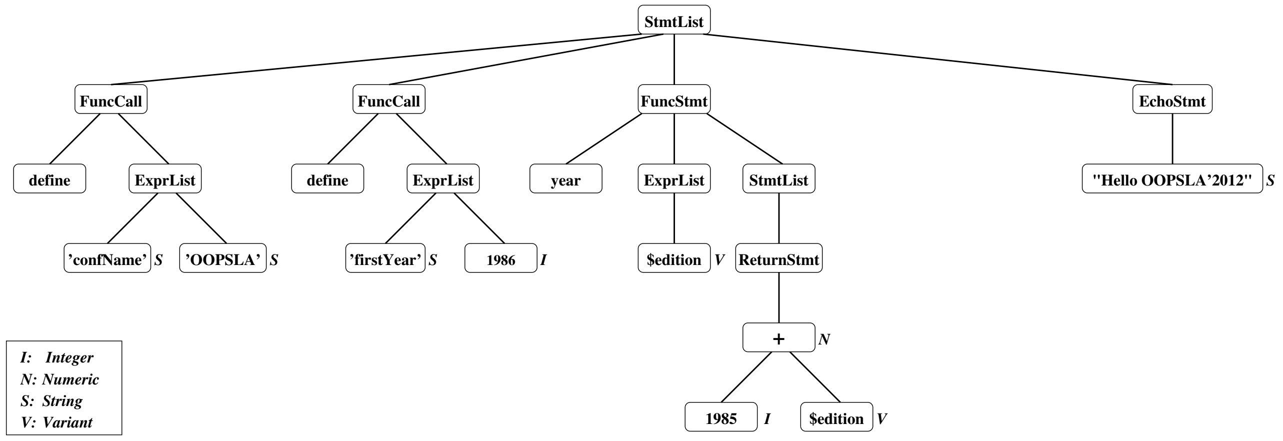


Table of Constants

Name	Type	Value
confName	String	'OOPSLA'
firstYear	Integer	1986

(c) Initial AST generated by the parser and constants collected by the analysis phase



(d) AST after pre-optimizer and type-inference phases

Figure 3. Example of HipHop compilation phases. (a) input PHP code; (b) sketch of the generated C++ code; (c) initial AST; (d) AST after pre-optimizer and type-inference phases.

ferred, HipHop uses the specific inferred types, for which the runtime system contains fast, specialized implementations.

In Figure 3(d), the AST nodes are annotated with the types inferred by HipHop. For example, in function `year`, although one of the operands of the addition has type *Variant*, the compiler infers that the result of the addition must be of *Numeric* type, and so does the return value of the function.

Phase 5 – Post-optimizer

After type inference, HipHop performs a couple of optimizations that benefit from type information, including algebraic and logical simplifications. In addition, another round of some pre-optimizer's optimizations (e.g. string concatenation, dead-code elimination) is performed to handle opportunities exposed by type-specific optimizations. In the example from Figure 3(d), the code has been fully optimized by the pre-optimizer, so there are no further opportunities left for the post-optimizer.

Phase 6 – Code Generation
The last step of HipHop is to traverse the AST and emit C++ code. The choice of C++ was made due to a few factors. First, C++ is known as one of the fastest languages and it has highly optimized compilers. HipHop has been using GCC [9] to compile the generated code, thus leveraging the substantial amount of optimizations that GCC implements. Second, C++ is a very flexible language, which makes it a good target for auto-generated code. Finally, C++ shares many similarities with PHP, including object orientation, which facilitate the translation process.

HipHop generates four groups of C++ files:

1. Class Header Files: Each PHP class has one C++ header file, which is included by other files that use that particular PHP class.
2. PHP-File Header Files: Each PHP file has one corresponding C++ header file with the prototype of functions that it defines. A PHP `include` statement is translated into a C++ `include` for the corresponding C++ header file. This effectively makes the functions defined in that file visible to the other files that include it.
3. Implementation Files: These files have implementation of converted PHP functions, class methods, and top-level statements (i.e. statements outside any function or method). Each of these C++ files is generated from one or more PHP files.
4. System Files: These files do not have corresponding PHP files and they are generated to store system-level functions and global lookup tables (discussed in Section 3.3).

Given the similarities between C++ and PHP, most constructs can be easily translated into C++ from the AST. Some notable exceptions include:

1. Some statements need to have a declaration added to header files as discussed above. These statements include functions, methods, classes, interfaces, traits, class variables and constants.
2. Top-level statements are wrapped into what we call a *pseudo-main* function. A PHP `include` statement including a file with top-level statements will also result in a C++ call to its pseudo-main function to execute these statements.
3. Some expressions have dynamic evaluations that do not have a C++ counterpart. These include dynamic variable, function, and class names as illustrated in Section 2.1.3. How HipHop supports these features is discussed in Section 3.3 below.

Figure 3(b) presents a sketch of the C++ file generated for the PHP script from Figure 3(a). The top-level `echo` statement is wrapped into the pseudo-main function `pm_php$hello_oopsla_php`. The `echo` function is provided by the HipHop runtime system.

3.3 Supporting PHP Semantics

In this section, we describe how HipHop handles some of the most dynamic and challenging PHP features discussed in Section 2.1.

3.3.1 Dynamic Typing

Support for dynamic typing is implemented with a combination of type inference (see Section 3.2) and a runtime system that implements the PHP operators for the variant type (at minimum) and other common types (for performance).

3.3.2 Dynamic Symbol Tables

In order to support dynamic name binding (Section 2.1.2) and resolution (Section 2.1.3), HipHop keeps track of global state during runtime in a *global symbol table* (GST). This table includes entries for:

- Global variables;
- Dynamically declared constants;
- Static variables inside functions, methods, and classes;
- Whether a file has been included or not;
- *Redeclared* functions and classes.

For functions and classes that are defined multiple times in the PHP code, called *redeclared*, HipHop generates all their versions (with unique names) and the GST keeps track of which version is dynamically included.

In scopes with symbol names that cannot be resolved statically (e.g. those in Section 2.1.3), HipHop also generates a *local symbol table* (LST). All variable accesses within such scopes then need to go through a helper that looks up the symbol in LST and GST. The LST is also used in the presence of PHP `extract` and `compact` functions.[2]

Finally, to support dynamic properties (see Section 2.1.3), HipHop keeps inside the objects a *property symbol table* (PST) that maps property names to values. The PST is used when accessing dynamic properties.

With this approach based on dynamic symbol tables, HipHop supports dynamic name binding and resolution in PHP (at a performance cost) while still leveraging static binding to generate efficient code for the most common cases.

3.3.3 Invocation Tables

HipHop uses a few *invocation tables* for invoking functions, instantiating objects, or including source files whose names cannot be resolved statically. In these situations, HipHop produces code that does a runtime lookup into the appropriate table, which is very similar to how Zend processes

[2] PHP's `extract` function imports the variables from a given array into the current local symbol table, while `compact` does the opposite.

function calls. HipHop's invocation tables map names of functions, classes, or files to the corresponding implementation. These invocation tables, which contain all known functions, classes, and files, is completely generated at compilation time, which is possible due to HipHop's whole-program analysis.

3.3.4 Volatile Symbols

In HipHop, we call the PHP symbols whose presence affect the execution *volatile symbols*. Section 2.1.4 illustrates how a program can check for the presence of functions and classes. In PHP's interpreter-based reference implementation, the symbols are only present after they are dynamically loaded, which occurs when the files containing their definitions are included. With HipHop, all symbols are naturally loaded statically, before the program starts execution. In the cases where the dynamic-loading semantics is relevant, HipHop provides that illusion in the following way. For each symbol that requires this behavior, HipHop keeps a flag that tells whether the symbols has been loaded or not. This flag is initially unset, and HipHop emits code to set it at the program point where the symbol is declared. The real difficulty with this approach is to determine which symbols are volatile, and thus need to be tracked by this mechanism. To address this problem, HipHop uses a few heuristics (e.g., `foo` is marked volatile if it appears in `class_exists('foo')`). However, precisely determining whether a symbol is volatile may ultimately depend on program inputs and dynamic execution flow, and thus it is an undecidable problem. Therefore, besides its heuristics, HipHop also leverages a user-provided list of symbols that need to be treated as volatile.

3.3.5 Reflection

Contrary to PHP, C++ does not have any support for reflection. To support PHP's reflection APIs, HipHop statically generates a set of tables containing the necessary data, including: class information, function prototypes, and source location information. The source location information for functions also enables generating a PHP-level stack trace from the C++ frames on the stack.

4. Evaluation

This section evaluates HipHop and compares it with other PHP execution engines. The evaluation was performed on 64-bit Linux servers based on Intel® Xeon processors. We used the latest release of HipHop at the time of this writing, and Zend PHP version 5.3.10, which was released in February 2012.

We present experiments running both standard benchmarks and the Facebook web site. The benchmarks include the widely used `bench.php` provided by Zend, and also the set of PHP applications from the computer language benchmarks [22].

4.1 Standard Benchmarks

Figure 4 presents the performance comparison for our set of benchmarks. The first portion of these programs (up to `strcat`) is part of Zend's `bench.php`, while the remaining ones are from the computer language benchmarks [22]. The bars in Figure 4 are the execution times relative to Zend. HipHop is faster than Zend in all benchmarks. The maximum speedup is on `simpleudcall`, for which HipHop is $260\times$ faster than Zend. This program simply has a loop calling an empty function, which HipHop inlines to eliminate the call overhead. The geometric mean shows that HipHop is $5.6\times$ faster than Zend for this set of benchmarks.

In addition to Zend and HipHop, Figure 4 also presents results for the HHVM interpreter (HHVMi) [7]. HHVMi is a PHP bytecode interpreter recently developed at Facebook. In principle, HHVMi is similar to Zend (also a bytecode interpreter), but it uses the HipHop extensions, runtime, and web server. Figure 4 shows that HHVMi is on average 17.9% slower than Zend and $6.3\times$ slower than HipHop.

Notice that none of these benchmarks use the PHP features unsupported by HipHop (described in Section 3.1). This fact confirms the assumption made in HipHop that such features are uncommon (and sometimes even insecure).

4.2 Facebook Workloads

In this section, we present experimental results of running the Facebook web site with production traffic, i.e. serving real user requests.

Facebook currently uses HipHop to compile all its PHP code base. Most of HipHop's compilation phases have been parallelized, which enables HipHop to compile Facebook's huge code base from PHP into C++ in 3.5 minutes using a single 12-core server. The generated C++ code, which is about $5\times$ bigger than the original PHP as measured in lines of code, is then compiled in parallel using GCC and distcc on a cluster of servers. This step takes 8 minutes. The final binary is linked using Google's Gold linker, and it is distributed to all Facebook web servers using bittorrent.

Unfortunately, due to new language extensions in HipHop that are widely used in the Facebook code base (e.g. richer type hints and yield generators), Zend PHP has been unable to run the Facebook web site since August 2010. Given this limitation, we present two sets of experiments to give an idea of HipHop's current benefits over Zend running Facebook production traffic. The first set of experiments is based on historical improvements made to HipHop since August 2010, and the second one uses HHVMi.

4.2.1 Tracking HipHop's Historical Improvements

In these experiments, we measured the initial performance improvement of HipHop over Zend when Zend was last able to run the Facebook web site. From that point, we kept track of performance improvements made to HipHop over itself. For that, a baseline HipHop branch was created

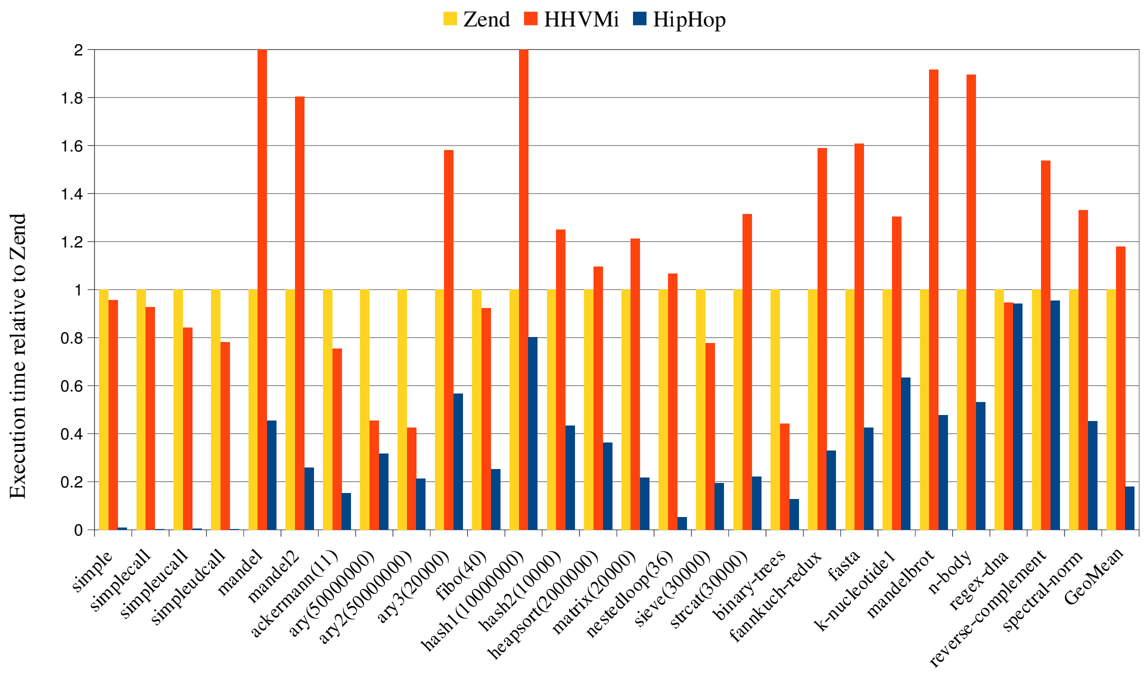


Figure 4. Performance comparison of Zend, HHVMi, and HipHop using standard PHP and computer language benchmarks

and updated with the minimal changes necessary to keep it running the site. These updates included support for new language extensions, but no performance enhancements.

In August 2010, we last compared the performance of HipHop and Zend running Facebook production traffic. At that time, the version of Zend used was 5.1.3. For this comparison, two clusters of servers were used, one with HipHop and another with Zend, and load balancers ensured that the two sets received equal traffic. The average CPU time serving requests in each cluster was computed across a large period of time. On average, HipHop completed web requests 2.5× faster than Zend. Since HipHop was much faster, the servers in its cluster were underutilized. We also compared the throughput of the clusters when the servers were fully loaded. To do so, we configured the load balancers to provide enough traffic to keep the CPUs in both clusters 100% utilized. The results of this experiment, measured in number of requests-per-second (RPS), demonstrated that the HipHop cluster achieve throughput 2.5× higher than Zend. On the left side of Figure 5, this throughput data is plotted relative to HipHop.

Figure 5 also plots the throughput improvements made to HipHop after August 2010. This comparison was performed using the same methodology described above, except that the Zend cluster was changed to use the baseline HipHop branch instead. The load balancers were again set to keep the servers in each cluster fully utilized, so that we could measure the maximum throughput achieved by each server. Overall, HipHop's throughput over this period improved by another 2.3×. Although we cannot directly compare with Zend running the current Facebook web site, these data suggest that HipHop's throughput is now 5.8 (2.5 × 2.3) times higher than what Zend 5.1.3 would achieve. These improvements resulted from additional tuning, static analyses, and optimizations described in Section 3 that were added in this period.

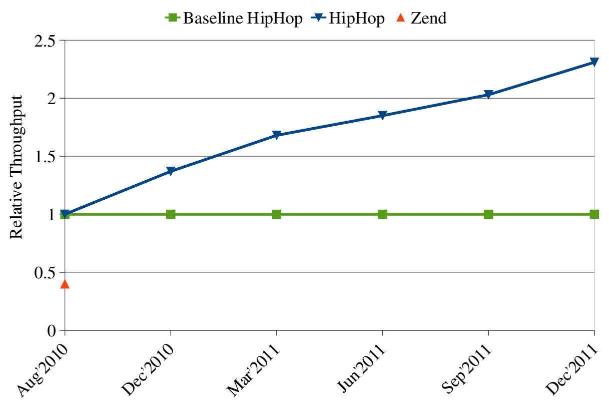


Figure 5. HipHop throughput over time, running Facebook production traffic. The baseline is the throughput of HipHop's version from August 2010, which is when Zend last ran Facebook.

Given the nature of the Facebook application, which processes huge data sets, one could expect Facebook's web servers to be I/O bound. However, given the great amount of work done on data fetching and caching, Facebook's web servers are actually CPU bound. This is the reason why HipHop greatly impacts the performance of Facebook's web servers, in a similar fashion to how it also improved the performance of benchmark applications.

4.2.2 HipHop vs. HHVMi

As another experiment, we compared the performance of HipHop against HHVMi [7], which is a PHP bytecode interpreter akin to Zend. Although HHVMi's performance is slightly lower than Zend on the benchmarks evaluated in Section 4.1, a comparison between HipHop and HHVMi effectively isolates the core differences between the two different PHP execution engines. This is because HHVMi uses the same HipHop extensions, runtime, and web server.

Following the same methodology described in the previous section, we ran HHVMi and HipHop in production, serving normal Facebook web requests. This experiment demonstrated that, on average, HipHop served web requests 5.5× faster than HHVMi.

Overall, our evaluation demonstrates that HipHop is more than 5× faster than other PHP implementations. This has been confirmed through extensive experiments both using standard PHP benchmarks and running the Facebook web site in production. Our findings are also aligned with what has been reported by another company that converted from Zend to HipHop and published an experimental evaluation [11].

5. Related Work

The continuous expansion of the Internet and adoption of PHP for server-side development has motivated a number of efforts to improve PHP performance. We briefly describe various of these efforts in this section.

One approach to improve the performance of PHP has been to leverage Java Virtual Machines (JVM) by translating PHP into Java bytecode [16, 17, 21]. Tatsubori et al. [21] have pursued this approach by retrofitting PHP into IBM's P9 JVM. Compared to Zend, their evaluation shows a 20-30% improvement on SPECweb2005, and 2.5×-9.5× improvement on micro-benchmarks. A similar approach has been followed by Quercus [17] and Project Zero [16]. Quercus's reported performance is similar to Zend with bytecode caching (APC) [1, 17]. For Project Zero, performance has not been reported and the emphasis is on the interoperability between PHP and Java.

Analogous to the JVM-based efforts above, Phalanger [2] compiles PHP to Microsoft CIL bytecode. Here again, Phalanger's emphasis has been more on interoperability than performance. Phalanger's recently reported performance is on par with Zend PHP using bytecode caching [13].

Another JIT-based project for PHP is the HappyJIT [10], which is based on PyPy [19]. On `bench.php` and the computer language benchmarks, HappyJIT's performance compared to Zend varies from 4× slower to 8× faster.

Most closely to HipHop's approach, a couple of other projects have compiled PHP to C, including `phc` [3] and Roadsend [20]. Unlike HipHop, `phc` makes extensive calls to the Zend runtime and does not exploit many of the opportunities available through static compilation. For example, it does not leverage static binding and, instead, generates code that performs extensive dynamic lookups. Furthermore, `phc` does not perform static analysis and optimizations, and its benefits over Zend come mostly from avoiding the dispatch overheads in the interpreter loop. Its reported speedup over Zend on `bench.php` is 1.53× [3]. Roadsend followed a similar approach, but it is more simplistic and does not support various PHP extensions. Recent evaluations show that Roadsend is not able to run all `bench.php` and that its performance is similar to Zend [10].

Lemos [12] presents a performance comparison among HipHop, `phc`, and Zend running `bench.php`. Although this evaluation did not use the latest HipHop release, it still found HipHop to be the fastest engine by far, being 3× faster than the second one (`phc`).

6. Conclusion

Although scripting languages have considerably increased in popularity over the past decades, their performance is still a main blocker for building large, CPU-intensive systems. This paper described how the HipHop compiler addresses the performance problem of a typical and widely used scripting language, PHP. The key for achieving performance in HipHop is the use of static compilation, to generate very efficient code whenever possible, combined with a set of techniques to support various inherently dynamic PHP features. Overall, our thorough experimental evaluation demonstrated that HipHop is on average 5.5× more efficient than traditional PHP execution engines based on interpreters. The drastic efficiency gains provided by HipHop have resulted in its adoption not only inside Facebook but also in a number of other Internet companies. Finally, we believe that the approach and techniques used to make HipHop a success can also be applied to other scripting languages. As such, we expect HipHop to have great impact in improving the performance of scripting languages in general, and thus help expand their uses.

Acknowledgments

For their support, feedback on the HipHop compiler project, and/or feedback on earlier drafts of this paper, we thank: Ali-Reza Adl-Tabatabai, Aditya Agarwal, Albert Bergman, Ervan Darnell, Jordan DeLong, Steven Grimm, Andrei Homescu, Harry Li, Scott MacFiggen, David Mortenson, Mike Paleczny, Jay Parikh, Girish Patangay, Daniel Pepper, David Reiss, Sonya Rikhtverchik, Jeff Rothschild, Paul Saab, Mike Schroepfer, and the rest of the HipHop team at Facebook.

We also thank the anonymous reviewers for their valuable feedback.

References

[1] APC: Alternative PHP Cache. Web site: http://php.net/manual/en/book.apc.php.

[2] 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.

[3] 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.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28:331–388, March 2006.

[6] Facebook, Inc. The HipHop compiler for PHP. Available at: https://github.com/facebook/hiphop-php/wiki/.

[7] Facebook, Inc. The HipHop Virtual Machine. Web site: https://www.facebook.com/note.php?note_id=10150415177928920, December 2011.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. 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.

[9] GCC. Gnu Compiler Collection. Web site: http://gcc.gnu.org.

[10] 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.

[11] Hyves, Inc. HipHop for PHP at Hyves. Web site: http://hyvesblogonproductdevelopment.blogspot.com/2011/10/hiphop-for-php-at-hyves.html, October 2011.

[12] M. Lemos. PHP compiler performance. Available at: http://www.phpclasses.org/blog/post/117-PHP-compiler-performance.html, February 2010.

[13] J. Misek. Improved wordpress performance with phalanger. Web site: http://www.php-compiler.net/blog/2011/phalanger-wordpress-performance, 2011.

[14] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31:23–30, March 1998.

[15] PHP eval. Web site: http://php.net/manual/en/function.eval.php.

[16] Project Zero. Web site: https://www.projectzero.org/php/.

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

[18] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, pages 52–78, 2011.

[19] 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.

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

[21] 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 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121–132, 2010.

[22] The Computer Language Benchmarks Game. Web site: http://shootout.alioth.debian.org/.

[23] S. Warner and J. Worley. SPECweb2005 in the real world: Using IIS and PHP. In *Proceedings of SPEC Benchmark Workshop*, 2008.

[24] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 278–287, 2010.

[25] Zend PHP. Web site: http://php.net.