A Practical Solution for Scripting Language Compilers

Paul Biggar, Edsko de Vries, David Gregg

Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland

Abstract

Although scripting languages are becoming increasingly popular, even mature scripting language implementations remain interpreted. Several compilers and reimplementations have been attempted, generally focusing on performance.

Based on our survey of these reimplementations, we determine that there are three important features of scripting languages that are difficult to compile or reimplement. Since scripting languages are defined primarily through the semantics of their original implementations, they often change semantics between releases. They provide C APIs, used both for foreign-function interfaces and to write third-party extensions. These APIs typically have tight integration with the original implementation, and are used to provide large standard libraries, which are difficult to re-use, and costly to reimplement. Finally, they support run-time code generation. These features make the important goal of correctness difficult to achieve for compilers and reimplementations.

We present a technique to support these features in an ahead-of-time compiler for PHP. Our technique uses the original PHP implementation through the provided C API, both in our compiler, and in our generated code. We support all of these important scripting language features, particularly focusing on the correctness of compiled programs. Additionally, our approach allows us to automatically support limited *future* language changes. We present a discussion and performance evaluation of this technique.

Key words: Compiler, Scripting Language

1. Motivation

Although scripting languages[1] are becoming increasingly popular, most scripting language implementations remain interpreted. Typically, these implementations are slow, between one and two orders of magnitude slower than C. There are a number of reasons for this. Scripting languages have grown up around interpreters, and were

Email addresses: `pbiggar@cs.tcd.ie` (Paul Biggar), `edsko.de.vries@cs.tcd.ie` (Edsko de Vries), `david.gregg@cs.tcd.ie` (David Gregg)

[1]We consider PHP, Perl, Python, Ruby and Lua as the current crop of scripting languages. We exclude Javascript since it does not share many of the attributes we discuss in this paper. Notably, it is standardized, and many distinct implementations exist, none of which are canonical.

generally used to glue together performance sensitive tasks. Hence, the performance of the language itself was traditionally not important. As they have increased in prominence, larger applications are being developed entirely in scripting languages, and performance is increasingly important.

The major strategy for retrofitting performance into an application written in a scripting language is to identify performance hot-spots, and rewrite them in C using a provided C API. Modern scripting languages are equipped with C APIs which can interface with the interpreter, in fact, in many cases the interpreters themselves are written using these APIs. Though this is not a bad strategy—it is a very strong alternative to rewriting the entire application in a lower level language—a stronger strategy may be to compile the entire application. Having a compiler automatically increase the speed of an application is an important performance tool, one that contributes to the current dominance of C, C++ and Java.

However, it is not straight-forward to write a scripting language compiler. The most important attribute of a compiler—more important than speed—is correctness, and this is difficult to achieve for a scripting language. Scripting languages do not have any standards or specifications[2]. Rather, they are defined by the behaviour of their initial implementation, which we refer to as their "canonical implementation"[3]. The correctness of a later implementation is determined by its semantic equivalence with this canonical implementation. It is also important to be compatible with large standard libraries, written in C. Both the language and the libraries often change between releases, leading to not one, but multiple implementations with which compatibility must be achieved.

In addition, there exist many third-party extensions and libraries in wide use, written using the language's built-in C API. These require a compiler to support this API in its generated code, since reimplementing the library may not be practical, especially if it involves proprietary code.

A final challenge is that of run-time code generation. Scripting languages typically support an **`eval`** construct, which executes source code at run-time. Even when **`eval`** is not used, the semantics of some language features require some computation to be deferred until run-time. A compiler must therefore provide a run-time component, with which to execute the code generated at run-time.

In `phc` (5), our ahead-of-time compiler for PHP, we are able to deal with the undefined and changing semantics of PHP by integrating the PHP code. At compile-time, we use the PHP system as a language oracle, giving us the ability to automatically adapt to changes in the language, and allowing us avoid the long process of documenting and copying the behaviour of myriad different versions of the language. We also generate C code which interfaces with the PHP system via its C API. This allows our compiled code to interact with built-in functions and libraries, saving not only the effort of reimplementation of large standard libraries, but also allowing us to interface

[2]This is becoming less true for Python and Lua, which now provide reference manuals.

[3]A canonical implementation differs subtly from a reference implementation, in that a reference implementation provides an implementation of a specification, while a canonical implementation provides the specification.

with both future and proprietary libraries and extensions. Finally, we reuse the existing PHP system to handle run-time code generation, which means we are not required to provide a run-time version of our compiler, which can be a difficult and error-prone process.

Since many of the problems we discuss occur with any reimplementation, whether it is a compiler, interpreter or JIT compiler, we shall generally just use the term 'compiler' to refer to any scripting language reimplementation. We believe it is obvious when our discussion only applies to a compiler, as opposed to a reimplementation which is not a compiler.

In Section 2.1, we provide a short motivating example, illustrating these three important difficulties: the lack of a defined semantics, emulating C APIs, and supporting run-time code generation. In Section 3, we examine a number of previous scripting language compilers, focusing on important compromises made by the compiler authors which prevent them from correctly replicating the scripting languages they compile. Our approach is discussed in Section 4, explaining how each important scripting language feature is correctly handled by re-using the canonical implementation. Section 5 discusses PHP's memory model. Section 6 discusses the complementary approach of using a JIT compiler. An experimental evaluation of our technique is provided in Section 7, including performance results, and supporting evidence that a large number of programs suffer from the problems we solve.

2. Challenges to Compilation

There are three major challenges to scripting languages compilers: the lack of a defined semantics, emulating C APIs, and supporting run-time code generation. Each presents a significant challenge, and great care is required both in the design and implementation of scripting language compilers as a result. We begin by presenting a motivating example, before describing the three challenges in depth.

2.1. Motivating Example

Listing 1 contains a short program segment demonstrating a number of features which are difficult to compile. The program segment itself is straight-forward, loading an encryption library and iterating through files, performing some computation and some encryption on each. The style uses a number of features idiomatic to scripting languages. Though we wrote this program segment as an example, each important feature was derived from actual code we saw in the wild.

Lines 3-6 dynamically load an encryption library; the exact library is decided by the `$engine` variable, which may be provided at run-time. Line 9 creates an array of hexadecimal values, to be used later in the encryption process. Lines 12-16 read files from disk. The files contain data serialized by the **`var_export`** function, which converts a data-structure into PHP code which when executed will create a copy of the data-structure. The serialized data is read on line 16, and is deserialized when line 17 is executed. Lines 20-28 represent some data manipulation, with line 20 performing a hashtable lookup. The data is encrypted on line 31, before being re-serialized and written to disk in lines 34 and 35 respectively. Line 37 selects the next file by incrementing the string in `$filename`.

```
define(DEBUG, "0");

# Create instance of cipher engine
include 'Cipher/' . $engine . '.php';
$class = 'Cipher_' . $engine;
$cipher = new $class();

# Load s_box
$s_box = array(0x30fb40d4, ..., 0x9fa0ff0b);

# Load files
$filename = "data_1000";
for($i = 0; $i < 20; $i++)
{
  if(DEBUG) echo "read serialized data";
  $serial = file_get_contents($filename);
  $deserial = eval("return $serial;");

  # Add size suffix
  $size =& $deserial["SIZE"];
  if ($size > 1024 * 1024 * 1024)
    $size .= "GB";
  elseif ($size > 1024 * 1024)
    $size .= "MB";
  elseif ($size > 1024)
    $size .= "KB";
  else
    $size .= "B";

  # Encrypt
  $out = $cipher->encrypt($deserial, $s_box);

  if(DEBUG) echo "reserialize data";
  $serial = var_export($out, 1);
  file_put_contents($filename, $serialized);

  $filename++;
}
```

Listing 1: PHP code demonstrating dynamic, changing or unspecified language features.

2.2. *Undefined Language Semantics*

A major problem for reimplementations of scripting languages is the languages' undefined semantics. Jones (14) describes a number of forms of language specification. Scripting languages typically follow the method of a "production use implementation" in his taxonomy. In the case of PHP, Jones says:

> The PHP group claim that they have the final say in the specification of PHP. This group's specification is an implementation, and there is no prose specification or agreed validation suite. There are alternate implementations [...] that claim to be compatible (they don't say what this means) with some version of PHP.

As a result of this lack of abstract semantics, compilers must instead adhere to the concrete semantics of the canonical implementation for correctness. However, different releases of the canonical implementation may have different concrete semantics. In fact, for PHP, changes to the language definition occur as frequently as a new release of the PHP system. In theory, the language would only change due to new features. However, new features frequently build upon older features, occasionally changing the original semantics. Older features are also modified with bug fixes. Naturally, changes to a feature may also introduce new bugs, and there exists no validation suite to prevent these bugs from being considered features. In a number of cases we have observed, a "bug" has been documented in the language manual, and referred to as a feature, until a later release when the bug was fixed. As a result of these changes, even the same feature in different versions of the language may have different semantics.

While in a standardized language, like C or C++, the semantics of each feature is clearly defined[4], in a scripting language, the task of determining the semantics can be arduous and time consuming. Even with the source code of the canonical implementation available, it is generally impossible to guarantee that the semantics are copied exactly.

2.2.1. *Literal Parsing*

A simple example of a change to the language is a bug fix in PHP version 5.2.3, which changed the value of some integer literals. In previous versions of PHP, integers above `LONG_MAX`[5] were converted to floating-point values—unless they were written in hexadecimal notation (e.g. 0x30fb40d4). In this case, as in our example on line 9 of Listing 1, they were to be truncated to the value of `LONG_MAX`. Since version 5.2.3, however, these hexadecimal integers are converted normally to floating-point values.

2.2.2. *Libraries*

One of the major attractions of scripting languages is that they come "batteries included", meaning they support a large standard library. However, unlike the C++ or

[4]Standardized languages also consider some semantics 'undefined', meaning an implementation can do anything in this case. No scripting language features are undefined, since they all do *something* in the canonical implementation.

[5]Constant from the C standard library representing the maximum signed integer representable in a machine word.

Java standard libraries, a scripting language's standard library is typically written in C, using the C API. Compilers which do not emulate the C API must instead reimplement the libraries. Since the libraries are not specified, they are liable to change, and new libraries are constantly being added.

2.2.3. Built-in Operators

The lack of abstract semantics also means that it is difficult to know the exact behaviour of some language constructs, especially due to PHP's weak-typing. Addition, for example, is more general in PHP than in C. Its behaviour depends on the run-time type of the operands, and overflows integers into floats[6]. There is a significant amount of work in determining the full set of semantics for each permutation of operator and built-in type. What, for example, is the sum of the string "hello" and the boolean value **`true`**[7]? As another example, the two statements `$a = $a + 1;` and `$a++;` are not equivalent. The latter will "increment" strings, increasing the ASCII value of the final character, another unlikely language feature, as shown in Listing 1 on line 37.

Truth is also complicated in PHP, due to its weak-typing rules. Conditional statements implicitly convert values to booleans, and the conversions are not always intuitive. Example of false values are `"0"`, `""`, `0`, **`false`** and `0.0`. Examples of true values are `"1"`, `1`, **`true`**, `"0x0"` and `"0.0"`.

2.2.4. Language Flags

In PHP, the semantics of the language can be tailored through use of the `php.ini` file. Certain flags can be set or unset, which affect the behaviour of the language.

The `include_path` flag affects separate compilation, and alters where files can be searched for to include them at compile time. The `call_time_pass_by_ref` flag decides whether a caller is permitted to pass its actual parameter to a function by reference, potentially overriding the function's default of passing by copy.

2.3. C API

A scripting language's C API provides its foreign-function interface. Typically, it is used for embedding the language into an application, creating extensions for the language, and writing libraries. A discussion of the merits of various scripting languages' C APIs is available (20).

Typically, the C API is the only part of the language with stable behaviour. Though features are added over time, the C API is in such heavy use that regressions and bugs are noticed quickly. We have seen that even when changes to the language and its libraries are frequent, changes to the behaviour of the C API are not.

2.4. Run-time Code Generation

A number of PHP's dynamic features allow source code, constructed at run-time, to be executed at run-time. Frequently these features are used as quick hacks, and they

[6]Feeley discusses (8) a similar problem in Scheme, in that several Scheme compilers incorrectly prevent integers from overflowing into *Bignum*s for performance reasons.

[7]An integer `1`, it seems.

are also a common vector for security flaws. However, there are a sufficient number of legitimate uses of these features that a compiler must support them.

2.4.1. Eval Statements

As demonstrated in Listing 1, the **`eval`** statement executes arbitrary fragments of PHP code at run-time. It is passed a string of code, which it parses and executes in the current scope, potentially defining functions or classes, calling functions whose names are passed by the user, or writing to user-named variables.

2.4.2. Include Statements

The PHP **`include`** statement is used to import code into a given script from another source file. Although similar in theory to the **`eval`** statement, this feature is generally used by programmers to logically separate code into different source files, in a similar fashion to C's *`#include`* directive, or Java's **`import`** declaration. However, unlike those static approaches, an **`include`** statement is executed at run-time, and the included code is only then inserted in place of the **`include`** statement.

Dynamic **`include`** statements are commonly used in PHP to provide a plugin facility, or to implement localization. In Section 7.5, we provide statistics about usage of dynamic and static includes (as well as **`eval`** statements) from a large number of publicly available PHP programs.

2.4.3. Variable-variables

PHP variables are simply a map of strings to values. Variable-variables provide a means to access a variable whose name is known at run-time—for example, one can assign to the variable `$x` using a variable containing the string value `"x"`. Access to these variables may be required by **`eval`** or **`include`** statements, and so this feature may take advantage of the infrastructure used by these functions. Variable functions are also accessible in this way, and Listing 1 shows a class initialized dynamically in the same manner.

3. Related Work

Having discussed the typical scripting language features, we examine previous scripting language compilers, discussing how they handled the challenging features in their implementations. We believe that many of their solutions are sub-optimal, either requiring great engineering or sacrifices which limit the potential speed improvement of their approach.

3.1. Undefined Semantics

The most difficult and rarely addressed issue is ensuring that a program is executed correctly by a reimplementation of a scripting language. In particular, it is rarely mentioned that different versions of a scripting language can have different semantics, especially in standard libraries.

Very few scripting language compilers provide any compatibility guarantees for their language. Instead, we very often see laundry lists of features which do not work, and libraries which are not supported. A number of implementations we surveyed chose to rewrite the standard libraries. UCPy (3), a reverse-engineered Python compiler, reports many of the same difficulties that motivated us: a large set of standard libraries, a language in constant flux, and a manual whose contents surprise its own authors. They chose to rewrite the standard library, even though it was 71,000 lines of code long, risking potential semantic differences with the official distribution.

Both Roadsend (24) and Quercus (22)—PHP compilers, referred to by Jones' quote in Section 2.2—reimplement a very small portion of the PHP standard libraries. In Shed Skin (6, Sect. 4.3.3), a Python-to-C++ compiler, the authors were unable to analyse or reuse Python's comprehensive standard library. Instead, library functions they wanted to support were both reimplemented in C++ and seperately modelled in Python.

Jython (16) and JRuby (15) are reimplementations of Python and Ruby, respectively, on the JVM. They reimplement their respective standard libraries in their respective host languages, and do not reuse the canonical implementation. A much better approach is employed by Phalanger (4, Sect. 3), a PHP compiler targeting the *.NET* run-time. It uses a special manager to emulate the PHP system, through which they access the standard libraries through the C API. They report that they are compatible with the entire set of extensions and standard libraries. However, Phalanger does not use the PHP system's functions for its built-in operators, instead rewriting them in its host language, `C#`. Many of PHP's most difficult features to compile involve its built-in operators, and we believe that reimplementing them is costly and error-prone.

In terms of language features, none of the compilers discussed have a strategy for automatically adapting to new language semantics. Instead, each provides a list of features with which they are compatible, and the degree to which they are compatible. None mentioned the fact that language features change, or that standard libraries change, and we cannot find any discussion of policies to deal with these changes.

A few, however, mention specific examples where they were unable to be compatible with the canonical implementation of their language. Johnson et al. (13) attempted to reimplement PHP from public specifications, using an existing virtual-machine. They reported problems caused by PHP's call-by-reference semantics. In their implementation, callee functions are responsible for copying passed arguments, but no means was available to inform the callee that an argument to the called function was passed-by-reference[8]. Shed Skin (6) deliberately chose to use restricted language semantics, in that it only compiles a statically-typed subset of Python.

However, two approaches stand out as having taken approaches which can guarantee a strong degree of compatibility. 211 (1) converts Python virtual machine code to C. It works by pasting together code from the Python interpreter, which corresponds to the bytecodes for a program's hot-spots. 211 is a compiler which is very resilient to changes in the language, as its approach is not invalidated by the addition of new opcodes. It's approach is more likely to be correct than any other approach we mention, including our own, though it comes at a cost, which we discuss is Section 7.2.

Python2C (25, Section 1.3.1) has a similar approach to `phc`, and, like both `phc`

[8] In PHP, call-by-reference parameters can be declared at function-definition time or at call-time.

and 211, provides great compatibility. Unfortunately, it comes with a similar cost to 211, as detailed in Section 7.2.

Pyrex (7) is a domain-specific language for creating Python extensions. It extends a subset of Python with C types and operations, allowing mixed semantics within a function. It is then compiled, in a similar fashion to our approach. Though they omit much of the language, it is easy to see that by following this approach, they have to ability to have a very high degree of compatibility with Python, even as the language changes.

3.2. C API

Very few compilers attempt to emulate the C API. However, Johnson et al. (13) provide a case study, in which they determine that it is not possible in their implementation, claiming that the integration between the PHP system and the extensions was too tight. We have also observed this, as the C API is very closely modelled on the PHP system's implementation. Phalanger (4) does not emulate the C API, but it does provide a bridge allowing programs to call into extensions and libraries. Instead of a C API, it provides a foreign-function interface through the .Net run-time. Jython (16) and JRuby (15) provide a foreign-function interface through the JVM, in a similar fashion.

3.3. Run-time Code Generation

A number of compilers (24; 13; 4; 15; 16) support run-time code generation using a run-time version of their compiler. Some (6; 22) choose not to support it at all. Quercus (22) in particular claims not to support it for security reasons, as run-time code generation can lead to code-injection security vulnerabilities. We show in Section 7.5 that this results in a large number of PHP programs which could not be run using the Quercus compiler.

While providing a run-time portion of the compiler is sensible for a JIT compiler, which has already been designed as a run-time system, most of these implementations are not JITs. However, providing this run-time portion requires that the implementation is suitable for run-time use; it must have a small footprint, it cannot leak memory, it must be checked for security issues, and it must generate code which interfaces with the code which has already been generated. These requirements are not trivial, and we believe the approach we outline in Section 4 affords the same benefits, at much lower engineering cost. we discuss using a JIT compiler in more detail in Section 6.

3.4. Other Approaches

Walker's optimizing compiler for Icon (29) uses the same system for its compiled code as its interpreter used. In addition, since they were in control of both the compiler and the run-time system, they modified the system to generate data to help the compiler make decisions at compile-time. Typically, scripting language implementations do not provide a compiler, and compilers are instead created by separate groups. As a result, it is generally not possible to get this tight integration, though it would be the ideal approach.

4. Our Approach

Nearly all of these approaches have been deficient in some manner. Most were not resilient to changes in their target language, and instead reimplemented the standard libraries (16; 15; 24; 22; 3; 13; 6). Those which handled this elegantly still failed to provide the C API (4), and those which achieved a high degree of compatibility (7; 25; 1) failed to provide a means to achieving good performance.

In `phc`, our ahead-of-time compiler for PHP, we are able to correct all of these problems by integrating the PHP system into both our compiler and compiled code. At compile-time, we use the PHP system as a language oracle, allowing us to automatically adapt to changes in the language, and saving us the long process of documenting and copying the behaviour of many different versions of the language. Our generated C code interfaces with the PHP system at run-time, via its C API. This allows our compiled code to interact with built-in functions and libraries and to re-use the existing PHP system to handle run-time code generation.

4.1. Undefined Semantics

4.1.1. Language Semantics

One option for handling PHP's volatile semantics is to keep track of changes in the PHP system, with separate functionality for each feature and version. However, our link to the PHP system allows us to resiliently handle both past and future changes.

For built-in operators, we add calls in our generated code to the built-in PHP function for handling the relevant operator. As well as automatically supporting changes to the semantics of the operators, this also helps us avoid the difficulty of documenting the many permutations of types, values and operators, including unusual edge cases.

We solve the problem of changing literal definitions by parsing the literals with the PHP system's interpreter, and extracting the value using the C API. If the behaviour of this parsing changes in newer versions, the PHP system's interpreter will still parse it correctly, and so we can automatically adapt to some language changes which have not yet been made.

We handle language flags by simply querying them via the C API. With this, we can handle the case where the flag is set at configure-time, build-time, or via the `php.ini` file. No surveyed compiler handles these scenarios.

4.1.2. Libraries and Extensions

One of the largest and most persistent problems in creating a scripting language reimplementation is that of providing access to standard libraries and extensions. We do not reimplement any libraries or extensions, instead re-using the PHP system's libraries via the C API. This allows us to support proprietary extensions, for which no source code is available, which is not possible without supporting the C API. It also allows support for libraries which have yet to be written, and changing definitions of libraries between versions.

4.2. C API

Naturally, we support the entire C API, as our generated code is a client of it. This goes two ways, as extensions can call into our compiled code in the same manner as the code calls into extensions.

Integrating the PHP system into the compiler is not complicated, as most scripting languages are designed for embedding into other applications (20). Lua in particular is designed expressly for this purpose (12). In the case of PHP, it is a simple process (11) of including two lines of C code to initialize and shutdown the PHP system. We then compile our compiler using the PHP "embed" headers, and link our compiler against the "embed" version of `libphp5.so`, the shared library containing the PHP system.

Users can choose to upgrade their version of the PHP system, in which case `phc` will automatically assume the new behaviour for the generated code. However, compiled binaries may need to be re-compiled, since the language has effectively changed.

4.3. Run-time Code Generation

In addition to being important for correctness and reuse, the link between our generated code and the PHP system can be used to deal with PHP's dynamic features, in particular, the problem of run-time code generation.

Though the **`include`** statement is semantically a run-time operation, `phc` supports a mode in which we attempt to include files at compile-time, for performance. Since the default directories to search for these files can change, we use the C API to access the `include_path` language flag. If we determine that we are unable to include a file, due to its unavailability at compile-time, or if the correctness of its inclusion is in doubt, we generate code to invoke the interpreter at run-time, which executes the included file. We must therefore accurately maintain the program's state in a format which the interpreter may alter at run-time. Our generated code registers functions and classes with the PHP system, and keeps variables accessible via the PHP system's local and global run-time symbol tables. This also allows us support variable-variables and the **`eval`** statement with little difficulty.

4.4. Compiling with phc

`phc` parses PHP source code into an *Abstract Syntax Tree*(5) from which C code is generated. The generated code interfaces with the PHP C API, and is compiled into an executable—or a shared library in the case of web applications—by a C compiler. Listings 2–5 show extracts of code compiled from the example in Listing 1. In each case, the example has been edited for brevity and readability, and we omit many low-level details from our discussion.

Listing 2 shows the `main()` method for the generated code. `phc` compiles all top-level code into a function called `__MAIN__`. All functions compiled by `phc` are added to the PHP system when the program starts, after which they are treated no differently from PHP library functions. To run the compiled program, we simply start the PHP system, load our compiled functions, and invoke `__MAIN__`.

Listing 3 shows a simple assignment. Each value in the PHP system is stored in a `zval` instance, which combines type, value and garbage-collection information. We access the `zvals` by fetching them by name from the local symbol table. We then

```
int main (int argc, char *argv[]) {
  php_embed_init (argc, argv);
  php_startup_module (&main_module);
  call_user_function ("__MAIN__");
  php_embed_shutdown ();
}
```

Listing 2: `phc` generated code is called via the PHP system.

```
zval* p_i;
php_hash_find (LOCAL_ST, "i", 5863374, p_i);
php_destruct (p_i);
php_allocate (p_i);
ZVAL_LONG (*p_i, 0);
```

Listing 3: `phc` generated code for `$i = 0;`

carefully remove the old value, replacing it with the new value and type. We use the same symbol tables used within the PHP system, with the result that the source of the `zval`—whether interpreted code, libraries or compiled code—is immaterial.

```
static php_fcall_info fgc_info;
php_fcall_info_init (
  "file_get_contents", &fgc_info);

php_hash_find (
  LOCAL_ST, "f", 5863275, &fgc_info.params);

php_call_function (&fgc_info);
```

Listing 4: `phc` generated code for `file_get_contents($f);`

Listing 4 shows a function call. Compiled functions are accessed identically to library or interpreted functions. The function information is fetched from the PHP system, and the parameters are fetched from the local symbol table. They are passed to the PHP system, which executes the function indirectly.

```
php_file_handle fh;
php_stream_open (Z_STRVAL_P (p_TLE0), &fh);
php_execute_scripts (PHP_INCLUDE, &fh);
php_stream_close (&fh);
```

Listing 5: `phc` generated code for `include($TLE0);`

Listing 5 shows an **`include`** statement. The PHP system is used to open, parse, execute and close the file to be included. The PHP system's interpreter uses the same symbol tables, functions and values as our compiled code, so the interface is seamless[9].

4.5. Optimizations

The link to the C API also allows `phc` to preform a number of optimizations, typically performing computation at compile-time, which would otherwise be computed at run-time.

4.5.1. Constant-folding

The simplest optimization we perform is constant folding. In Listing 1, line 23, we would attempt to fold the constant expression `1024 * 1024` into `1048576`. PHP has five scalar types: booleans, integers, strings, reals and nulls, and 18 operators, leading to a large number of interactions which need to be accounted for and implemented. By using the PHP system at compile-time, we are able to avoid this duplicated effort, and stay compatible with changes in future versions of PHP. We note that the process of extracting the result of a constant folding does not change if the computation overflows.

4.5.2. Pre-hashing

We can also use the embedded PHP system to help us generate optimized code. Scripting languages generally contain powerful syntax for hashtable operations. Listing 1 demonstrates their use on line 20.

When optimizing our generated code, we determined that 15% of our compiled application's running time was spent looking up the symbol table and other hashtables, in particular calculating the hashed values of variable names used to index the local symbol table. However, for nearly all variable lookups, this hash value can be calculated at compile-time via the C API, removing the need to calculate the hash value at run-time. This can be seen in Listing 3, when the number `5863374` is the hashed value of `"i"`, used to lookup the variable `$i`. This optimization removes nearly all run-time spent calculating hash values in our benchmark.

4.5.3. Symbol-table Removal

In Section 4.3, we discussed keeping variables in PHP's run-time symbol tables. This is only necessary in the presence of run-time code generation. If we statically guarantee that a particular function never uses run-time code generation—that is to say, in the majority of cases—we remove the local symbol table, and access variables directly in our generated code.

4.5.4. Pass-by-reference Optimization

PHP programs tend to make considerable use of functions written in the C API. Since functions may be called which are not defined at compile-time, we must add run-time checks to determine whether parameters should be passed by reference or by copy. However, we are able to query the function's signatures of any function written in the C API, which allows us to calculate these at compile-time, rather than run-time.

[9] We note that the seamless interface requires being very careful with a `zval`'s reference count.

4.5.5. Caching function calls.

Since PHP is a dynamic language, with functions only defined at run-time, we must lookup functions by name before we can call them. Initially, we began by looking up a function each time we called it. However, since functions cannot change their definition after they are first defined, we cache the function lookup after the first time we call it. This speedup from this optimization is significant (around 23% compared with a similar version of `phc` without this optimization).

4.6. Caveats

Our approach allows us to gracefully handle changes in the PHP language, standard libraries and extensions. Clearly though, it is not possible to automatically deal with large changes to the language syntax or semantics. When the parser changes—and it already has for the next major version of PHP—we are still required to adapt our compiler for the new version manually. Though we find it difficult to anticipate minor changes to the language, framing these problems to use the PHP system is generally straight-forward after the fact. Finally, we are not resilient to changes to the behaviour of the C API; empirically we have noticed that this API is very stable, far more so than any of the features implemented in it. This is not assured, as bugs could creep in, but these tend to be found quickly since the API is in very heavy use, and we have experienced no problems in this regard.

5. Interactions with the PHP Memory Model

When assessing the performance of a programming language implementation, it is natural to think that most of the execution time is likely to be spent performing computations. In fact, as we discuss in Section 7.1, the run-time system often has a major impact on performance. This is particularly true for scripting languages for three main reasons. First, scripting languages generally provide automatic memory management to reclaim objects that are no longer in use. The memory manager adds to execution time, whether it uses a tracing garbage collector, or as in the case of PHP, reference counting. Secondly, even scalar values in scripting languages are typically implemented with data-structures rather than simple C scalars, because additional information such a type and memory management information must be stored along with the value. Thirdly, the main data-structuring feature provided by scripting language is the table, which is typically implemented using hashtables. Thus, even simple record or array type data-structures need a more complicated memory representation, which often consists of more than one single piece of memory. For these reasons, to optimize the performance of a compiler which uses the canonical implementation, it is essential to understand the memory model used by the implementation.

In this section, we discuss the *PHP memory model* and pitfalls which occur when linking to such a model.

5.1. The PHP Memory Model

The primitive unit of data in PHP is the `zval`, a small structure encompassing a union of values—objects, arrays and scalars—and memory-management counters and flags. A PHP variable is a symbol-table entry pointing to a `zval`, and multiple variables can point to the same `zval`, using reference counting for memory management.

PHP assignment is by copy, meaning that semantically the l-value becomes a copy of the r-value. This is not only true of scalars: PHP arrays are deeply copied during an assignment, and object references are copied to a new run-time `zval`. As an optimization, the PHP system causes the l-value to share the r-value's `zval`, increasing its reference count. The variables are said to become part of the same *copy-on-write* set. Thus, even though an assignment is semantically a copy, the assigned value is shared until it is required to be altered.

Assignment can also be by reference, which puts the two variables in the same *change-on-write* set, in a similar fashion. This sets the `is_ref` flag of the shared `zval`, indicating that the variables in this set all reference each other. Setting a variable's value, where that variable is part of a change-on-write set, changes the value of all the other variables in that set.

Variables in a copy-on-write set share the same `zval`, but are not semantically related. Although this is an optimization applied by the PHP system, it is a feature which `phc` must deal with to interact with the PHP system, and so it reuses it for performance. In order to update the value of a variable in a copy-on-write set, it must first be *separated*. A copy of its `zval` is created—a deep copy in the case of arrays and strings—and the original `zval` has its reference count decremented. Variables in a change-on-write set must similarly be separated if they are assigned by copy.

Assignment to a variable in a change-on-write set overwrites the `zval`'s value field, changing the value of all the variables in that set. Variables with a reference count of one, which are in neither a copy-on-write or change-on-write set—are treated similarly.

The PHP interpreter keeps pointers to a variable's `zval` in global and function-local symbol-tables—hashtables indexed by the variable's name. When a function finishes execution, the local symbol-table is destroyed, decreasing the reference count of all `zval`s contained within. The global symbol-table is destroyed at the end of the execution of a script.

5.2. Pitfalls with the Memory System

In creating `phc`, we came across a number of pitfalls which we believe can affect any scripting language compiler. We describe the most important which we have come across[10]. At first, our naively generated code was around ten times slower than the PHP interpreter. This was primarily due to the fact that our code used significantly more memory than the PHP interpreter. The most important factor in this was our use of three-address code. In order to simplify our compiler transformations and code generation, we lowered complex expressions into three address code by adding assignment to temporary variables. However, these extra assignments increase the reference count of a `zval`, meaning not only that a program's memory remains live for a longer period, but also that there are more separations, leading to extra memory allocations, copying, and subsequent deallocations.

[10]Another interesting pitfall is described by Tozawa et al (28).

```
for ($i = 0; $i < $N; $i++)
{
  $str .= "hello";  // concat
}
```

Listing 6: String concatenation benchmark.

```
1 for ($i = 0; $i < $N; $i++)
2 {
3   $T1 = "hello";
4   $T2 = $str;         // T2.refcount++;
5   $T2 = $T2 . $T1;    // concat
6   $str = $T2;
7 }
```

Listing 7: Lowered string concatenation benchmark.

In a simpler language such as C, copying a value has no ramifications for the copied value, so introducing three-address code does not have great performance side-effects. However, in PHP, copying a value will increase its reference count, meaning it must be separated before it can be written to or altered. We removed many of the cases in which we generated poor code simply by being more careful during our conversion to three address code.

To highlight the magnitude of this problem, consider Listing 6. In this example, we accidentally turn an $O(N)$ algorithm into an $O(N^2)$ one, shown in Listing 7. This is a subtle, but interesting problem stemming from the interaction of 3AC and copy-on-write semantics. Other scripting languages which use copy-on-write, such as Perl and Tcl, may also experience this problem.

Listing 6 is a string concatenation benchmark, referred to later as *strcat*. The `.=` operator performs in-place concatenation, in this case appending `"hello"` onto the end of the string in `$str`. Though this code did not strictly need to be lowered to 3AC, our over-zealous lowering algorithm added extra temporaries into this code, resulting in Listing 7. Semantically, these perform the same operations. However, the `zval` pointed to by `$T2` has a reference count of two after line 4, meaning the string cannot be concatenated in place. Instead, `$T2` must be separated, even though it will be freed on line 4 of the next loop iteration.

It is interesting to observe the difference in performance between the two similar pieces of code. Listing 7 takes $O(N)$ time[11]. By contrast, in Listing 7, when `$str` must be copied in every iteration due to an increased reference count, the same work take $O(N^2)$ time in total. We note that this problem does not only occur due to 3AC. It is not always trivial to determine the reference count of a variable, and problems such

[11]We ignore the complexity of memory allocation due to increasing the size of the string, which will be the same in both cases.

as these may appear in user-code by accident.

6. Just-in-time Compilers

Just-in-time compilers (JITs) (2) are an alternative to interpreting or ahead-of-time compiling. In recent years, the growing popularity of managed languages running on virtual machines, such as Java's JVM and the Microsoft .Net framework, has contributed to the growth of JITs.

JIT compilers' optimizations are not inhibited by dynamic features, such as reflection and run-time code generation. Method specialization (23) compiles methods specifically for the actual run-time types and values. Other techniques can be used to gradually compile hot code paths (10; 31).

JITs, however, suffer from great implementation difficulty. They are typically not portable between different architectures, one of the great advantages of interpreters. Every modern scripting language's canonical implementation is an interpreter, and many implementations sacrifice performance for ease of implementation. The Lua Project (12, Section 2), for example, strongly values portability, and will only use ANSI C, despite potential performance improvement from using less portable C dialects, such as using computed `goto`s in GNU C.

In addition to being difficult to retarget, JIT compilers are difficult to debug. While it can be difficult to debug generated code in an ahead-of-time compiler, it is much more difficult to debug code generated into memory, especially when the JIT compiles a function multiple times, and replaces the previously generated code in memory. By contrast, our approach of generating C code using the PHP C API is generally very easy to debug, using traditional debugging techniques.

Much of the performance benefit of JIT compilers comes from inlining functions (26). However, scripting language standard libraries are typically written using the language's C API, not in the language itself, and so cannot be optimized using the JIT's inlining heuristics. We also expect a similar problem when current methods of trace JITs (9) are attempted to be ported from Javascript—in which entire applications are written mostly in Javascript—to other scripting languages. Achieving the kind of speeds achieved by Java JITs would require rewriting the libraries in the scripting language. As a result, it often takes great effort to achieve good performance in a JIT compiler. A prototype JIT for PHP (18) was recently developed using LLVM (17), but ran 21 times slower than the existing PHP interpreter.

7. Evaluation

7.1. PHP performance profile

Conventional wisdom states that a compiled program should run an order-of-magnitude faster than an interpreted program. In our experience, however, dynamic scripting languages do not follow this rule of thumb. Instead, a program written in a scripting language spends most of its run-time handling dynamic features, such as dynamic types and `zval`s. This limits the potential improvement of simply removing the interpreter loop. This is particularly important for a compiler like `phc` which re-uses the PHP

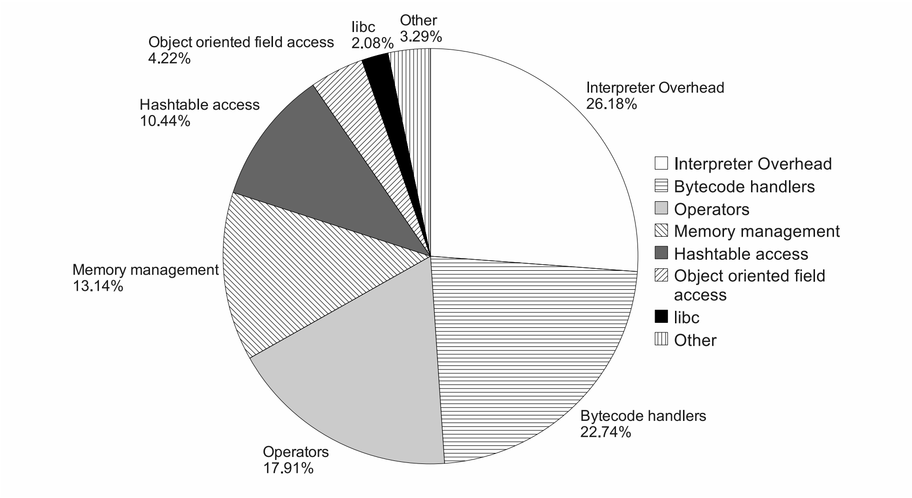


Figure 1: Profiling results of the PHP interpreter, using callgrind.

system, as many of the code paths executed will be the same, whether the program is interpreted or compiled.

To understand where time is spent in the PHP system, and to determine the potential speedup from optimization, we profiled the PHP system. Figure 1 shows the profile of a number of PHP benchmark applications, interpreted using PHP version 5.2.3, using the `callgrind` tool from valgrind 3.4.1 (21). We compiled PHP using `gcc` version 4.4.0, using the options `-O3 -g -NDEBUG`, targeting the x86-64 instruction set. We analysed the flat profile provided by callgrind, looking at the "*self*" results (that is, time spent in a function, not including time spent in the function's callees). We categorized each function in the profile into broad categories, based on our knowledge of the design of the PHP system.

Interpreter overhead includes time spent parsing, generating bytecode, running the interpreter loop and dispatching to bytecodes. *Bytecode handlers* are the code blocks dispatched to by the interpreter, which actually execute the desired operation. *Operators* includes time spent executing arithmetic and logical operators. *Memory management* is self explanatory. *Hashtable access* involves access to hashtables (which includes arrays, objects and symbol-tables), including calculating hash values from string keys. *Object oriented field accesses* excludes the actual hashtable access, but includes other object oriented overhead such as checking for special object oriented handlers. *libc* denotes time spent in the C standard library.

While there is a significant amount of time spent in interpreter overhead (26%), it is not nearly enough to allow for a order-of-magnitude speedup from compilation. This lends support to our approach, as compared to that of 211 and Python2C. Both of these Python compilers take a narrow approach, attempting only to remove interpreter overhead, but they do not allow for higher-level optimizations. This means that their techniques cannot achieve a great speedup if they were applied to the PHP system.

Nearly 18% of run-time time is spent performing calculations in the *Operators* category. This is principally due to PHP's dynamic typing. PHP uses opcodes which perform significantly more computation than, say, a Java bytecode. For example, an `add` uses a single opcode, like in Java. However, where a Java add opcode is little more than a machine add and an overflow check, PHP's add opcode calls an add function. This function, depending on the types of the operands, might merge two arrays, convert strings to integers, or a number of other operations.

We also see a 10% overhead due to hashtable accesses. Hashtables are used extensively in PHP, not only as the principle data-structure (as both arrays and associative arrays), but also to provide symbol-tables and objects. In theory, the PHP system's interpreter accesses every local variable through the local symbol-table. However, it uses an optimization similar to our symbol-table removal in Section 4.5.3, which prevents this overhead (19). As a result, all of the hashtable overhead comes from array manipulation, accesses to the global symbol-table, and accessing fields of objects.

PHP's dynamic typing cross-cuts all of these categories. Hashtables must be used in PHP's object orientation, as a result of objects' dynamic types. A great deal of memory management is due to allocating `zvals` for every value in the program, a result of dynamic typing. A lot of the overhead of operators are due to checking types before performing the computation, which might be cheap by comparison. Thus dynamic types not only take up time in the PHP system, but also prevent compiling PHP programs to more efficient representations. We expect that static analyses can be developed which can remove many of these type checks and allow more efficient compilation, which we intend to follow up on in future work.

7.2. Performance

The major motivation of this research is to demonstrate a means of achieving correctness in a scripting language reimplementation. However, we are also able to increase the performance of our compiled code, compared to the PHP system's interpreter.

The PHP designers use a small benchmark (27), consisting of eighteen simple functions, iterated a large number of times, to test the speed of the PHP interpreter.

We compared the generated code from `phc` with the PHP system's interpreter, version 5.2.3. We used Linux kernel version 2.6.29.2 on an Intel Xeon 5138 with four cores[12], rated at 2.13Ghz (clocked at 1.6 Ghz), with 12GB of RAM and a 1MB cache per CPU. Both our compiled code and the PHP system were compiled with `gcc` version 4.4.0, using `-O3 -NDEBUG` compiler flags.

Figure 2 shows the execution time of our generated code relative to the PHP interpreter. `phc` compiled code performs faster on 16 out of 18 tests. The final column is the arithmetic mean of the speedups, showing that we have achieved a total speedup of 1.55. In Figure 3, our metric is memory usage, measured using the *space-time* measure of Valgrind's (21) *massif* tool (version 3.2). Our graph shows the per-test relative

[12]Note that all of our benchmarks are single-threaded, and that PHP does not support threads at a language level.

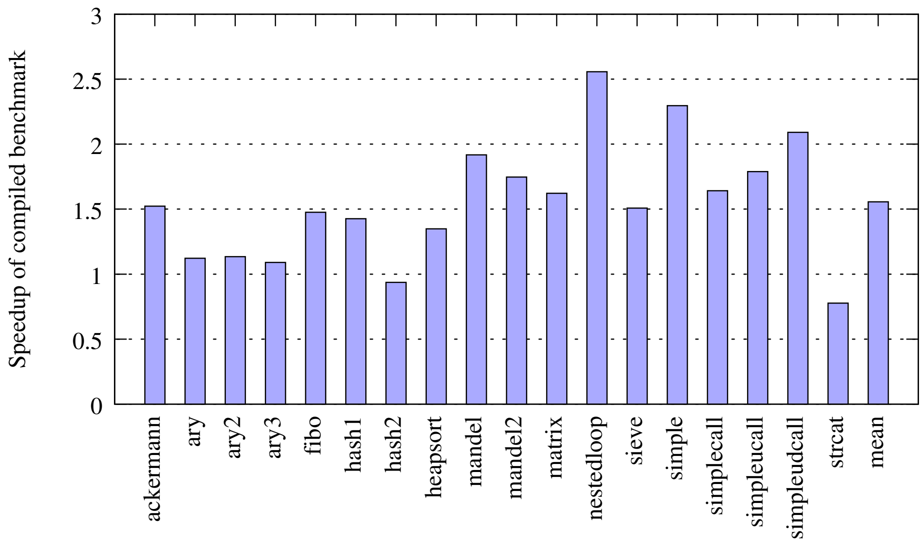


Figure 2: Speedups of `phc` compiled code vs the PHP interpreter. Results greater than one indicate `phc`'s generated code is faster than the PHP interpreter. The mean bar shows `phc`'s speedup of 1.55 over the PHP interpreter.

memory usage of one implementation over the other. The final column is the arithmetic mean of the reductions in memory usage, showing a reduction of 1.30.

It can be expected that we are able to optimize away the interpreter overhead, as discussed in Section 7.1, to achieve a speedup of 1.35. This is in the same league as previous implementations. Python2C (25, Section 1.3.1) is reputed to have a speedup of approximately 1.2, using a similar approach to ours, including some minor optimizations. 211 (1) only achieves a speedup of 1.06, the result of removing the interpreter dispatch from the program execution, and performing some local optimizations. It removes Python's interpreter dispatch overhead, and removes stores to the operand stack which are immediately followed by loads. We do not benefit from 211's optimization as peephole stack optimization will also not work for PHP, which does not use an operand stack.

However, our speedup is in some cases much greater than that which can be achieved by simply removing the interpreter overhead. In most cases, these are due to the optimizations which we discussed in Section 4.5. However, these are mitigated in some cases by poor code generation, especially related to hashtables, for example in `ary`, `ary2`, `ary3` and `hash2`. By contrast, we achieve much better speedups in functions which primarily manipulate loops and integers, in particular `nestedloop` and `mandel`.

We expect that traditional data-flow optimizations will also greatly increase our performance improvement, and our approach allows this in the future, which neither 211 nor Python2C allow. We believe that without this ability, 211 and Python2C are likely dead-ends, with their performance limited by their approaches.

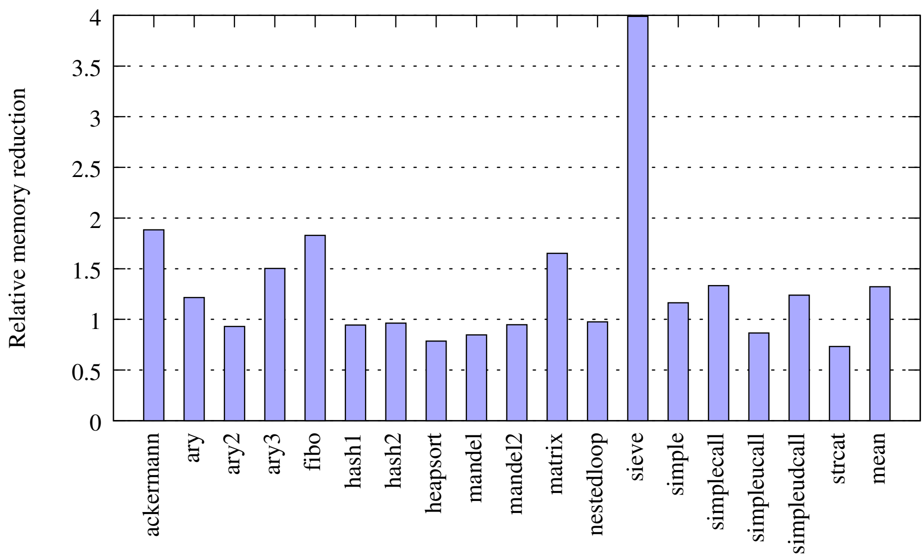


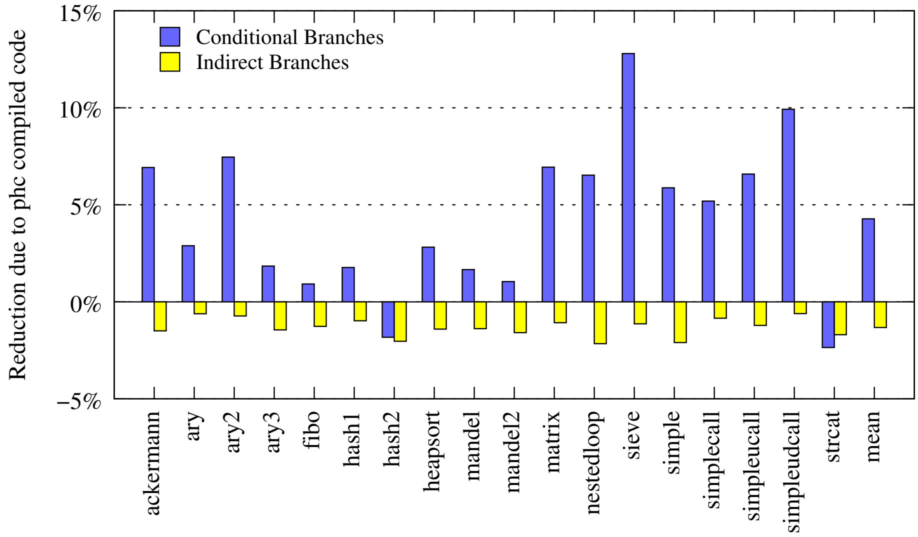
Figure 3: Relative memory usage of `phc` compiled code vs the PHP interpreter. Results greater than one indicate `phc`'s generated code uses less memory than the PHP interpreter. The mean bar shows `phc`'s a memory reduction of 1.30 over the PHP interpreter.

We also believe that the PHP system could achieve higher performance with a better implementation. However, the run-time work which slows PHP down also slows down our generated code, and so as PHP is improved, our speedup over PHP will likely remain constant.

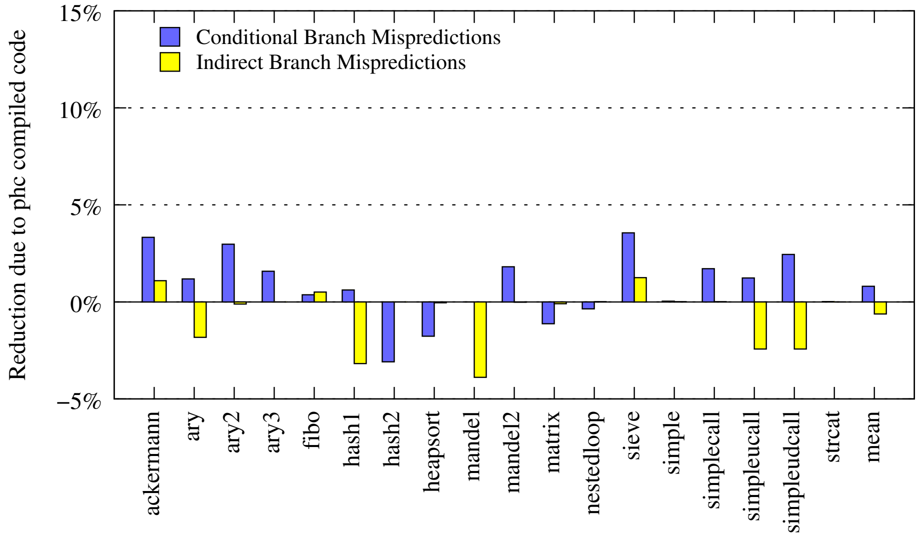
7.3. Performance examination

In order to understand why we achieved our performance improvement, we analysed both interpreted and compiled PHP benchmarks using the *cachegrind* tool from Valgrind 3.4.1, a hardware simulator. We measured a wide range of metrics including instruction counts, level-1 and level-2 data and instruction cache access, and branch behaviour. We use the same benchmarks, tools and program versions as discussed in Section 7.2.

Figure 4 presents branch prediction results, with Figure 4a showing the change in the number of branches, and Figure 4b showing the change in branch mispredictions. Results above zero indicate the decrease in branches as a percentage of instructions executed in the compiled program; results below zero indicate an increase. Figure 4b shows the difference in branch mispredictions scaled by the approximate cost of a branch misprediction. We choose 12 as this cost factor, roughly the length of a modern pipeline.



(a) Hardware simulation results comparing the number of branches in phc compiled code vs that of the PHP interpreter. Results are presented as a percentage of the instruction count. Results greater than zero indicate phc's generated code executes fewer branches.



(b) Hardware simulation results comparing the number of branch mispredictions in phc compiled code vs that of the PHP interpreter. The number of mispredictions are multiplied by 12 (representing a 12-stage pipeline), and presented as a percentage of the instruction count. Results greater than zero indicate phc's generated code generates fewer branch mispredictions.

Figure 4: Branch misprediction results.

A major difference between interpreters and compilers is that an interpreter loop typically leads to a great number of indirect branch mispredictions. Our results do not show this expected decrease in branch mispredictions however. Instead, we have a slight increase in indirect branches, of approximately 2%, and a larger decrease in conditional branches. In Figure 4b, we can see that the number of branch mispredictions does not decrease in our compiled code. It shows that even when scaled by a factor of 12, the cost of branch mispredictions costs no more than an equivalent of 1% of instructions executed. We also see that the slight increase in indirect branch mispredictions is mitigated by the slight decrease in conditional branch mispredictions.

We believe that the cost of the interpreter loop is not great in the PHP interpreter, when compared to the cost of dynamic features. Our generated code heavily uses switch statements in order to handle dynamic typing, and it appears that the reduction in the number of indirect mispredictions due to interpreter overhead is small compared mispredictions due to type checks. We speculate that the PHP interpreter more often uses conditional statements for the same purpose. Indeed, it appears that the overall number of branch mispredictions is largely the same in both compiled and interpreted programs.

We also measured changes in level-1 and level-2 cache misses, for both instruction and data caches. The difference in these misses is insignificant (that is, approaching 0%) when compared to instruction count, so we do not present them visually. We would expect to have in increase in instruction cache misses due to essentially inlining the bytecode handlers, but this did not materialize. We believe that with larger benchmarks, this may become more apparent.

It is clear that the speed of the running programs is not greatly affected by cache accesses or branch predictors. Figure 5 shows the decrease in instruction count and memory accesses due to compilation. Since the number of cache misses is not different, we surmise that the memory accesses removed due to compilation were level-1 cache hits, which have a low cost. Nevertheless, the ebb and flow of Figure 5 matches that of our speedup in Figure 2. It seems clear that the decrease in instruction count is due somewhat to the decrease in conditional branches. Indeed, in Figure 4a only two benchmarks (`hash2` and `strcat`) have an increase in conditional branches, and those same benchmarks are the only ones to result in a slowdown instead of a speedup in Figure 2.

As a result, we believe that our speedups come not from removing the cost of mispredictions in the interpreter loop, but instead through a combination of removing the rest of the interpreter overhead, and small optimizations. One of the costs of the interpreter is an extra layer of indirection when accessing `zval`s. While we store pointers to `zval`s in registers, the interpreter fetches pointers to `zval`s from memory, leading to increased memory accesses. While most of our simple optimizations are local, and aimed at reducing the instruction count, removing symbol-tables is aimed at reducing memory accesses, at which we appear to have been largely successful.

7.4. Feedback-directed optimization

Our technique is roughly similar to inlining the PHP system's bytecode handlers. In theory, this could allow the code to be rearranged based on feedback-directed optimization (FDO). This might allow the C compiler to do aggressive optimization, in

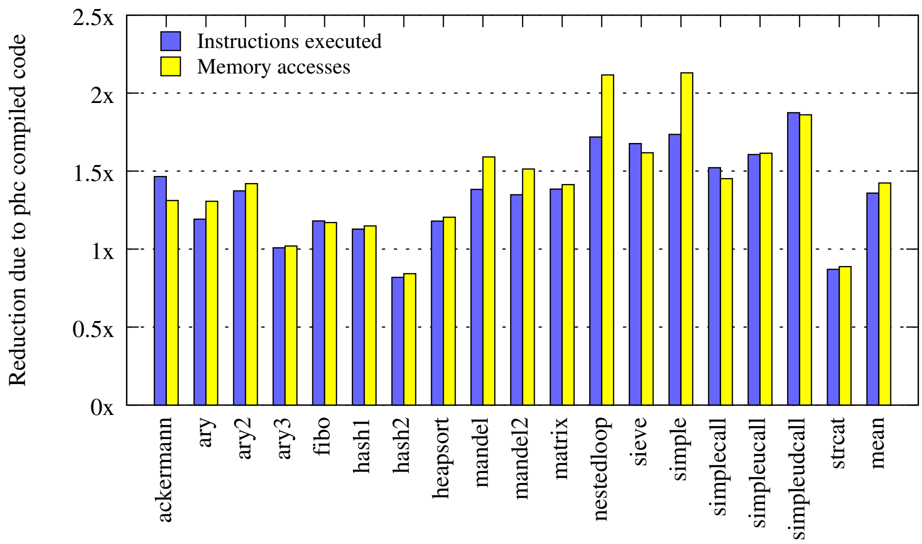


Figure 5: Hardware simulation results comparing the number of executed instructions and memory accesses in `phc` compiled code vs that of the PHP interpreter. Results greater than one indicate `phc`'s generated code performs better.

a similar technique to speculative inlining (8) or trace trees (9). Ideally, this would mitigate the slowdown of some of PHP's dynamic features, in particular its dynamic type checks, by moving the most likely code into a straight path, eliding pipeline stalls and branch mispredictions.

In order to determine whether such profiling has a beneficial effect, we reran our benchmarks using the `gcc` 4.4's FDO feature. Figure 6 shows the speed improvements over PHP 5.2.3, when using feedback directed optimization. PHP was configured as discussed above. We compiled `phc` generated code in the same manner as above, with the exception that we used the FDO options from `gcc` 4.4.0. We compiled the benchmarks initially using the `-fprofile-generate` flag. After running the generated executable, we compiled the benchmarks again using its feedback, with the `-fprofile-use -fprofile-generate` flags. Finally, we reused that feedback when compiling the benchmarks again using the `-fprofile-use` flag only.

In Figure 6, the "Without FDO" bar repeats the data from Figure 2. The "With FDO" bar shows the speedup over the PHP interpreter, when the code is compiled using FDO. Note that neither the PHP interpreter, nor the PHP system, are compiled using FDO.

It seems that while we achieve a small speedup from FDO, we are not able to automatically achieve large speedups. FDO causes our speedup to increase from 1.55 to 1.63. Most of the results indicate a small speedup, with the occasional small slowdown. While this average speedup is not insignificant, it is clear than most of the changes we seek can not be done at such a low-level, but will instead have to be handled within

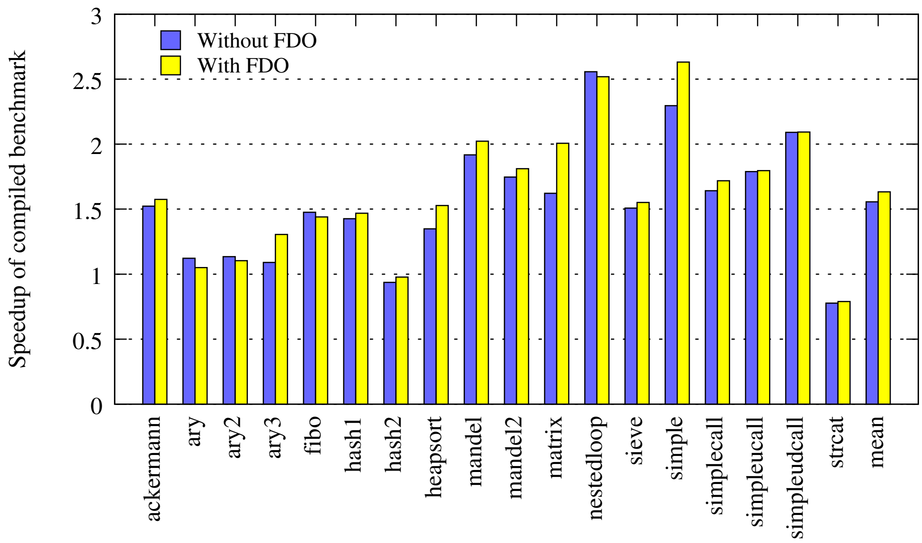


Figure 6: Speedups of `phc` compiled code vs the PHP interpreter, with and without FDO. Results greater than one indicate `phc`'s generated code is faster than the PHP interpreter. The "Without FDO" bar repeats the results from Figure 2. The mean "With FDO" bar shows `phc`'s speedup of 1.63 over the PHP interpreter, when using feedback-directed optimization.

`phc`. In the future, we will attempt to incorporate FDO within `phc`, applying a technique like that of Feeley (8).

Currently FDO provides a small speedup which is not possible in an interpreted environment. Our generated code separates the bytecode handlers' code paths in a context-sensitive manner. Since the C code is essentially inlined, it can be optimized using the profile for a single application. Naturally, we link the compiled code to the PHP system, which is not optimized in this way. However, we are still able to automatically achieve a small improvement by exposing `phc` generated code to the C compiler.

This optimization is not reasonable for an interpreted program. Other programs may need to be executed by the same interpreter, and may not benefit from the same optimizations, due to having a different profile.

7.5. Run-time code generation in PHP programs

The techniques we describe in this paper are particularly useful in the presence of run-time code generation. To evaluate its utility, we attempted to determine how often run-time code generation was used, by analysing a large number of publicly available PHP programs.

We automatically downloaded source code packages from the open-source code hosting site *sourceforge.net*. We selected packages which were labelled with the tag *"php"* and contained PHP source files. Of 645 packages chosen automatically, 581 of them contained an **include** statement. We consider these our test corpus, excluding packages without a single **include** statement. We believe files without **include** statements are likely to be simple programs or small classes, and are unlikely to be complete PHP programs. Figure 7 show overall statistics for the analysed code, showing we analysed over 42,000 files, incorporating over 8 million lines of code[13].

	PHP files	SLOC	**include**s
Total	42,523	8,130,837	66,999
Average	73	13,995	115

Figure 7: Package statistics for 581 PHP code packages, including number of files, number of source lines of code (SLOC), and number of **include** statements. **include** statements also includes **require**, **include_once** and **require_once** statements. "Average" means per package.

We created a plugin for the `phc` front-end to determine the presence of run-time code generation. We searched for either **eval** statements, or **include** statements which used dynamic features. We considered **include** statements which used only PHP constants, literal strings and concatenations to be static — all other features were deemed to be indicative of run-time code generation. We show the results of this analysis in Figure 8.

	Dynamic **include**s	**eval**s	Either	Neither
Instances	11,731	1,586		
Packages	331 (57%)	156 (26.9%)	358 (61.6%)	223 (38.4%)
Average	35.4	10.2		

Figure 8: Dynamic features in PHP code. The rows are: the number of instances of each feature, the number of packages using the feature at least once (with percentage of total packages), and the average number of times the feature is used by packages which use it.

From these figures, it is clear that support for run-time code generation is exceptionally important. It is used in 61% of PHP application, and when it is used, it is used extensively, with **eval**s appearing over 10 times in each package in which they appear, and dynamic includes appearing 35 times in each package in which they appear. This strongly indicates that our approach of supporting these features in our ahead-of-time compiler was wise, and that more static approaches would be unable to compile a large amount of PHP code. In fact, less than 39% of PHP applications do not use these dynamic features (though other dynamic features exist, which we did not attempt to detect).

Dynamic include statements are typically either plugin mechanisms or provide localisation. We suspect that in many cases, localization could be handled statically. This would mean searching for files in the source directories and replacing the dynamic include with a switch statement and a set of static includes. This approach is used in

[13]We measured lines of code using the Unix utility `wc`, so this figure includes blank lines and comments.

other tools (30). However, it is not safe, as the directory in which to search can be altered at run-time.

While dynamic includes are prevalent, and require special support, we note that the large majority of **`include`** statements use a static string. Of the 66,999 includes, fewer than 18% of them are dynamic. This implies that static analysis of PHP can be useful in a lot of cases, if code generation is not required.

8. Conclusion

Scripting languages are becoming increasingly popular, however, existing approaches to compiling and reimplementing scripting languages are insufficient. We present `phc`, our ahead-of-time compiler for PHP, which effectively supports three important scripting language features which have been poorly supported in existing approaches. In particular, we effectively handle run-time code generation, the undefined and changing semantics of scripting languages, and the built-in C API.

A principle problem of compiling scripting languages is the lack of language definition or semantics. We believe we are the first to systematically evaluate linking an interpreter—our source language's de facto specification—into our compiler, making it resilient to changes in the PHP language. We describe how linking to the PHP system helps to keep our compiler semantically equivalent to PHP, which has previously changed between minor versions.

We also generate code which interfaces with the PHP system. This allows us to reuse not only the entire PHP standard library, but also to invoke the system's interpreter to handle source code generated at run-time. We discuss how this allows us to reuse built-in functions for PHP's operators, replicating their frequently unusual semantics, and allowing us to automatically support those semantics as they change between releases. Changes to the standard libraries and to extensions are also supported with this mechanism.

Through discussing existing approaches, we show that our technique handles the difficulties of compiler scripting languages better than the existing alternatives. We show too that the percentage of PHP packages which benefit from our approach exceeds 60% of our sample. We show that we are able to achieve a speedup of 1.55 over the existing canonical implementation, and present a detailed discussion of why this is so.

Overall, we have shown that our approach is novel, worthwhile, and gracefully deals with a number of significant problems in compiling scripting languages, while maintaining semantic equivalence with the language's canonical implementation. We believe in the importance of correctness when compiling scripting languages, and that our research will provide the stepping stone on which future optimizations can be based.

Acknowledgements

The authors are indebted to Irish Research Council for Science, Engineering and Technology funded by the National Development Plan, whose funding made this work possible. We would also like to thank the SAC 2009 anonymous reviewers, whose comments helped us improve an earlier version of this paper.

References

[1] J. Aycock. Converting Python virtual machine code to C. In *Proceedings of the 7th International Python Conference*, 1998.

[2] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[3] J. Aycock, D. Pereira, and G. Jodoin. UCPy: Reverse engineering Python. In *PyCon DC2003*, March 2003.

[4] J. Benda, T. Matousek, and L. Prosek. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. In *.NET Technologies 2006*, May 2006.

[5] E. de Vries and J. Gilbert. Design and implementation of a PHP compiler front-end. Dept. of Computer Science Technical Report TR-2007-47, Trinity College Dublin, 2007.

[6] M. Dufour. Shed Skin: An optimizing Python-to-C++ compiler. Master's thesis, Delft University of Technology, 2006.

[7] G. Ewing. *Pyrex - a Language for Writing Python Extension Modules*. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

[8] M. Feeley. Speculative inlining of predefined procedures in an R5RS Scheme to C compiler. In *IFL*, pages 237–253, 2007.

[9] 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 *PLDI*, pages 465–478, 2009.

[10] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, New York, NY, USA, 2006. ACM.

[11] S. Golemon. *Extending and Embedding PHP*. Sams, Indianapolis, IN, USA, 2006.

[12] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, Jul 2005.

[13] G. Johnson and Z. Slattery. PHP: A language implementer's perspective. *International PHP Magazine*, pages 24–29, Dec 2006.

[14] D. M. Jones. Forms of language specification: Examples from commonly used computer languages. ISO/IEC JTC1/SC22/OWG/N0121, February 2008.

[15] JRuby [online]. `http://www.jruby.org`.

[16] Jython [online]. http://www.jython.org.

[17] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[18] N. Lopes. Building a JIT compiler for PHP in 2 days [online]. http://llvm.org/devmtg/2008-08/.

[19] S. Malyshev. Re: Compiled variables and backpatching, September 2007. http://news.php.net/php.internals/32460.

[20] H. Muhammad and R. Ierusalimschy. C APIs in extension and extensible languages. *Journal of Universal Computer Science*, 13(6):839–853, 2007.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[22] *Quercus: PHP in Java*. http://www.caucho.com/resin/doc/quercus.xtp.

[23] A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[24] Roadsend, Inc. *Roadsend PHP 2.9.x Manual*. http://code.roadsend.com/pcc-manual.

[25] M. Salib. Starkiller: A static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, 2004.

[26] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method in-lining for a Java just-in-time compiler. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 91–104, Berkeley, CA, USA, 2002. USENIX Association.

[27] The PHP Group. Zend benchmark [online]. http://cvs.php.net/viewvc.cgi/ZendEngine2/bench.php?view=co.

[28] A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. Copy-on-write in the PHP language. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 200–212, New York, NY, USA, 2009. ACM.

[29] K. Walker and R. E. Griswold. An optimizing compiler for the Icon programming language. *Softw. Pract. Exper.*, 22(8):637–657, 1992.

[30] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM Press.

[31] M. Zaleski, A. D. Brown, and K. Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, New York, NY, USA, 2007.