A Practical Solution for Scripting Language Compilers

Paul Biggar
pbiggar@cs.tcd.ie

Edsko de Vries
edsko.de.vries@cs.tcd.ie

David Gregg
david.gregg@cs.tcd.ie

Department of Computer Science
Trinity College Dublin
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 large standard libraries, which are difficult to re-use, and costly to reimplement. 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. Finally, they support run-time code generation. These features make the important goal of correctness difficult to achieve.

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 an 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, which has not previously been published.

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 generally used to glue together performance sensitive tasks. Hence, the performance of the language itself was 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. Rather, they are defined by the behaviour of their initial implementation, which we refer to as their "canonical implementation". 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 an 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 system—PHP's canonical implementation—into

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

[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.

both our compiler and compiled 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 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 an 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 the complementary approach of using a JIT compiler. The performance results of our compiler are presented in Section 6.

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`.

```
1  define(DEBUG, "0");
2
3  # Create instance of cipher engine
4  include 'Cipher/' . $engine . '.php';
5  $class = 'Cipher_' . $engine;
6  $cipher = new $class();
7
8  # Load s_box
9  $s_box = array(0x30fb40d4, ..., 0x9fa0ff0b);
10
11 # Load files
12 $filename = "data_1000";
13 for($i = 0; $i < 20; $i++)
14 {
15   if(DEBUG) echo "read serialized data";
16   $serial = file_get_contents($filename);
17   $deserial = eval("return $serial;");
18
19   # Add size suffix
20   $size =& $deserial["SIZE"];
21   if ($size > 1024 * 1024 * 1024)
22     $size .= "GB";
23   elseif ($size > 1024 * 1024)
24     $size .= "MB";
25   elseif ($size > 1024)
26     $size .= "KB";
27   else
28     $size .= "B";
29
30   # Encrypt
31   $out = $cipher->encrypt($deserial, $s_box);
32
33   if(DEBUG) echo "reserialize data";
34   $serial = var_export($out, 1);
35   file_put_contents($filename, $serialized);
36
37   $filename++;
38 }
```

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 [12] 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 the 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 interpreter. In theory, the language would only change due to new features. However, new features frequently build upon older features, changing the semantics of the older features. 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[2], 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`[3] 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 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. 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`**[4]? 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 can pass its actual parameter to a function by reference, potentially overriding the function's default of passing by copy.

[2]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.

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

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

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 [17].

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 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 immediately, 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.

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, requiring great engineering, or by making 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 compilers provide any compatibility guarantees for the language. Instead, we very often see laundry lists of features which do not work, and libraries which are not supported. A number of implementations [3, 6, 11, 13, 14, 19, 21] we surveyed chose to rewrite the standard libraries. UCPy, 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" [3]. They chose to rewrite the standard library, even though it was 71,000 lines of code long. Both Roadsend [21] and Quercus [19]—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, they had to both reimplement library functions they wanted to support in C++, and create a Python model for these functions to be supported by their compiler.

Jython [14] and JRuby [13] 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 the most difficult features of PHP 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. [11] 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[5]. 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 6.

Python2C [22, Section 1.3.1] has a similar approach to `phc`, and, like both `phc` and 211, provides great compatibility. Unfortunately, it comes with a similar cost to 211, as detailed in Section 6.

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.

[5]Call-by-reference parameters can be declared at function-definition time or at call-time.

3.2 C API

Very few compilers attempt to emulate the C API. However, Johnson et al. [11] 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 [14] and JRuby [13] provide a foreign-function-interface through the JVM, in a similar fashion.

3.3 Run-time Code Generation

A number of compilers [4, 11, 13, 14, 21] support run-time code generation using an run-time version of their compiler. Some [6,19] choose not to support it at all. Quercus [19] in particular claims not to support it for security reasons, as run-time code generation can lead to code-injection security vulnerabilities.

While providing an run-time portion of the compiler is sensible for a JIT compiler, which has already been designed as an 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.

3.4 Other Approaches

Walker's optimizing compiler for Icon [25] 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 typically created by separate groups. As a result, it is generally not possible to get this tight integration, though it would be the ideal approach.

In Section 5, we discuss using a JIT compiler, an alternate, and increasingly popular, method for compiling scripting languages.

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 [3,6,11,13,14,19,21]. Those which handled this elegantly still failed to provide the C API [4], and those which achieved a high degree of compatibility [1, 7, 22] 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 semantic changes in the PHP system, with handlers 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 [17]. Lua in particular is designed expressly for this purpose [10]. In the case of PHP, it is a simple process [9] 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 undecided, 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. The next section discusses this in greater detail.

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.

```
1 int main (int argc, char *argv[]) {
2   php_embed_init (argc, argv);
3   php_startup_module (&main_module);
4   call_user_function ("__MAIN__");
5   php_embed_shutdown ();
6 }
```

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

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__`.

```
1 zval* p_i;
2 php_hash_find (LOCAL_ST, "i", 5863374, p_i);
3 php_destruct (p_i);
4 php_allocate (p_i);
5 ZVAL_LONG (*p_i, 0);
```

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

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

```
1 static php_fcall_info fgc_info;
2 php_fcall_info_init (
3   "file_get_contents", &fgc_info);
4
5 php_hash_find (
6   LOCAL_ST, "f", 5863275, &fgc_info.params);
7
8 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.

```
1 php_file_handle fh;
2 php_stream_open (Z_STRVAL_P (p_TLE0), &fh);
3 php_execute_scripts (PHP_INCLUDE, &fh);
4 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[6].

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 4 primitive types: booleans, integers, strings and reals, 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 to stay compatible with changes in future versions of PHP. We note that the process of extracting the value after the 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 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 variable 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.

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

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 APIs are in very heavy use, and we have experienced no problems in this regard.

5. 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 contributes to the growth of JITs.

JIT compilers' optimizations are not inhibited by dynamic features, such as reflection and run-time code generation. Method specialization [20] compiles methods specifically for the actual run-time types and values. Other techniques can be used to gradually compile hot code paths [8, 26].

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 [10, 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 gotos 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 [23]. However, scripting language standard libraries are typically written using the language's C API, not in the language itself, and so cannot be analysed by the JIT's inlining heuristics. We also expect a similar problem when current methods of trace-JITs 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 [16] was recently developed using LLVM [15], but ran 21 times slower than the existing PHP interpreter.

6. PERFORMANCE EVALUATION

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 interpreter in the PHP system.

The PHP designers use a small benchmark [24], 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 interpreter, version 5.2.3. We used Linux kernel version 2.6.24-20 on an Intel Core 2 Duo[7], clocked at 2.13Ghz, with 2GB of RAM and a 2MB cache per CPU. The PHP system was compiled with `gcc` version 4.1, using `-O3`.

Figure 1a shows the execution time of our generated code relative to the PHP interpreter. `phc` compiled code performs faster on 15 out of 18 tests. The final column is the arithmetic mean of the speedups, showing that we have achieved a speed-up of 1.53. In Figure 1b, our metric is memory usage, measured using the *space-time* measure of the Valgrind [18] *massif* tool. Our graph shows the per-test relative memory usage of one implementation over the other. The final column is the arithmetic mean of the speedups, showing a reduction of 1.30.

When compared to a traditional compiler, a speed-up of 1.53 is modest. However, it is important to note that the majority of computation flows through the same paths as in the interpreted version. As with most interpreted scripting language implementations, little of the execution time is spent performing the actual computation. However, there are many more overheads than simply the cost of interpreter dispatch. The largest of these is the cost of the dynamic type system. We believe that we will be able to reduce this cost in the future, using static analysis and optimization techniques.

The PHP system's interpreter 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 function is little more than a machine add and an overflow check, the PHP add opcode calls an add function. This function, depending on the types of the operands, will merge two arrays, converting strings to integers, call a class method to convert an object to an integer, or any of a large number of different operations. As a result, removing the interpreter overhead does not lead to large performance benefits. From profiling the PHP system, approximately 15% of a program's execution time is due to interpreter overhead, including dispatch.

Our performance is very similar to that of 211 and Python2C. Python2C [22, Section 1.3.1] is reputed to have a speed-up of approximately 1.2, using a similar approach to ours, including some minor optimizations. 211 [1] only achieves a speed-up 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 the PHP system does not spend significant time performing interpreter dispatch. Peephole stack optimization will also not work for PHP, which does not use an operand stack.

However, we use a number of our own local optimizations, some of which were discussed in Section 4.5, which allow us the speed-up of 1.53. We showed that simply removing the interpreter overhead and compiling leads to no significant speed-up, and that our initial version, without optimizations, was much slower than the PHP system's interpreter. 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.

Python2C suffered from generating a very large amount of code, resulting in many instruction cache misses. Our implementation also suffers in the same fashion, but believe that with further analysis, the amount of generated C code can be greatly reduced, leading to greater speed improvement.

We also believe that PHP 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 speed-up over PHP will likely remain constant.

Importantly, we are able to get a large speed-up over the current implementation, while retaining the correctness a compiler requires, and preserving an avenue for future performance increases.

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

7. 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.

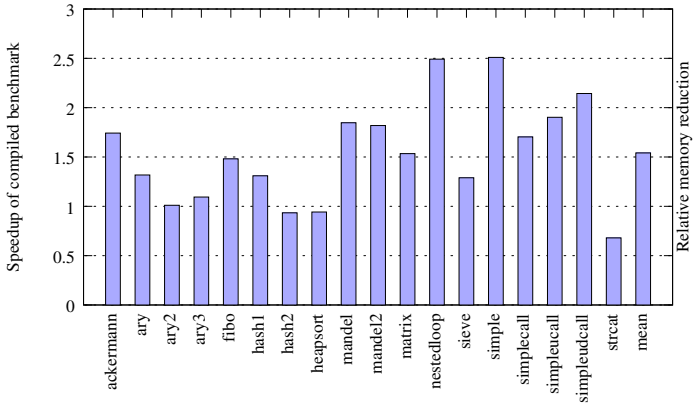
We also generate code which interfaces with the PHP system. This allows us to reuse not only the entire PHP standard library, but also to invoke the system's interpreter to handle source code generated at run-time. We discuss how this allows us to reuse built-in functions for PHP's operators, replicating 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 also show that we are able to achieve a speed-up of 1.53 over the existing implementation.

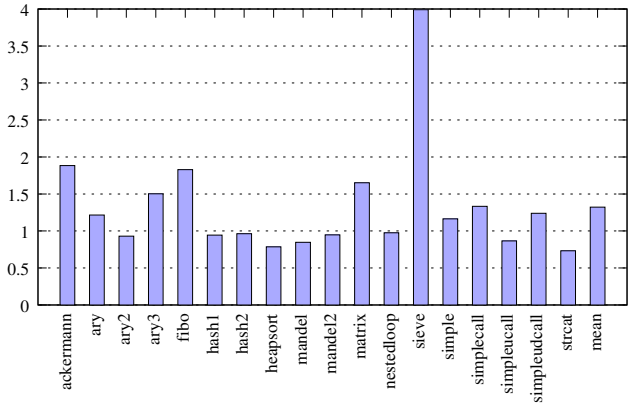
Overall, we have shown that our approach is novel, worthwhile, and gracefully deals with a number of significant problems in compiling scripting languages, while maintaining semantic equivalence with the language's canonical implementation. We believe in the importance of correctness when compiling scripting languages, and believe 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 paper's anonymous reviewers, whose comments helped us improve on earlier versions of this paper.



(a) 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 1.53 over the PHP interpreter.



(b) 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 memory reductions of 1.30 over the PHP interpreter.

Figure 1: Performance results.

8. 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] 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.

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

[10] 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.

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

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

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

[14] Jython [online]. `http://www.jython.org`.

[15] 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.

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

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

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

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

[20] 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.

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

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

[23] 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.

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

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

[26] 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.