HappyJIT: A Tracing JIT Compiler for PHP

Andrei Homescu
Department of Computer Science
University of California, Irvine
ahomescu@uci.edu

Alex Şuhan
Faculty of Mathematics and Computer Science
University of Bucharest
alex.suhan@gmail.com

Abstract

Current websites are a combination of server-generated dynamic content with client-side interactive programs. Dynamically - typed languages have gained a lot of ground in both of these domains. The growth of Web 2.0 has introduced a myriad of websites which contain personalized content, which is specific to the user. PHP or Python programs generate the actual HTML page after querying a database and processing the results, which are then presented by the browser. It is becoming more and more vital to accelerate the execution of these programs, as this is a significant part of the total time needed to present the page to the user.

This paper presents a novel interpreter for the PHP language written in RPython, which the PyPy translator then translates into C. The translator integrates into the interpreter a tracing just-in-time compiler which optimizes the hottest loops in the interpreted programs. We also describe a data model that supports all the data types in the PHP language, such as references and iterators. We evaluate the performance of this interpreter, showing that speedups up to a factor of 8 are observed using this approach.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, interpreters, optimization, run-time environments

General Terms Design, Languages, Performance

Keywords PHP, PyPy, RPython, just-in-time compilation, tracing, interpreter, dynamically typed languages

1. Introduction

In recent years we have seen a rapid explosion of Internet websites. While the earliest ones were completely static, written in HTML once and always presenting the same content to the user, later ones became more dynamic, showing personalized information specific to each user or visit, running various computations on data stored in some database. These computations have been split between server-side computations, where the server prepares a web page for each request specifically for that request, or client-side computations which are done on the user's machine. Various languages have grown very popular specifically for these purposes, such as Python and PHP on the server and JavaScript and ActionScript in the browser. Dynamic pages are today used in a variety of web pages, from search engines and social networking sites to online banking and shopping pages. Since the faster a page is generated on the server, the more requests from users can be served each second by that server, it is becoming vital to accelerate the execution of programs written in these languages.

One element that these languages have in common is their execution environment. A program written in one of these languages is then executed in a virtual machine, after being converted from its textual representation to some bytecode which is either written to disk and executed later (programs written in Java) or directly executed from memory (programs written in JavaScript and Python). Much research [15, 19, 20] has been done recently on improving the performance of the virtual machines running these programs. Prior research has shown that there are some techniques that have a great positive effect on performance, such as *just-in-time compilation*.

In this paper, we present the design and implementation of an optimized interpreter for one such language, PHP. We will show that on a significant portion of the tests in the PHPbench [4] and Computer Language Benchmarks Game [8] test suites, our implementation performs at least as well as the standard PHP interpreter and, in some cases, significantly better. This paper makes the following contributions:

- We present the use of PyPy to build a front-end for a language different from Python.
- We describe the design of a prototype PHP interpreter written in RPython.
- We present several optimizations to the data structures used by the interpreter.
- We compare the performance of our JIT-compiled interpreter with the Zend PHP engine.

2. Background

PHP is currently one of the most popular dynamically-typed languages, mostly due to its widespread use for generating of dynamic webpages. Whenever a user requests a page from a webserver, a PHP program generates the corresponding HTML response. In addition to its scripting features and support for many text-oriented algorithms, it also features many object-oriented and, in recent versions (the latest version as of writing this paper is PHP 5.3.6), functional language features, such as classes and lambdas. The official interpreter for this language is the Zend PHP engine [10].

Another widely used language is Python. Its official implementation uses an interpreter, although there has been much recent work into improving the performance of Python using *just-in-time compilation*, e.g., in the Unladen Swallow [9] using the LLVM framework [22], run-time type specialization in Psyco [25] or PyPy using a tracing JIT [15]. The latter is an implementation of Python written in Python itself. Figure 1 illustrates its high-level design. It

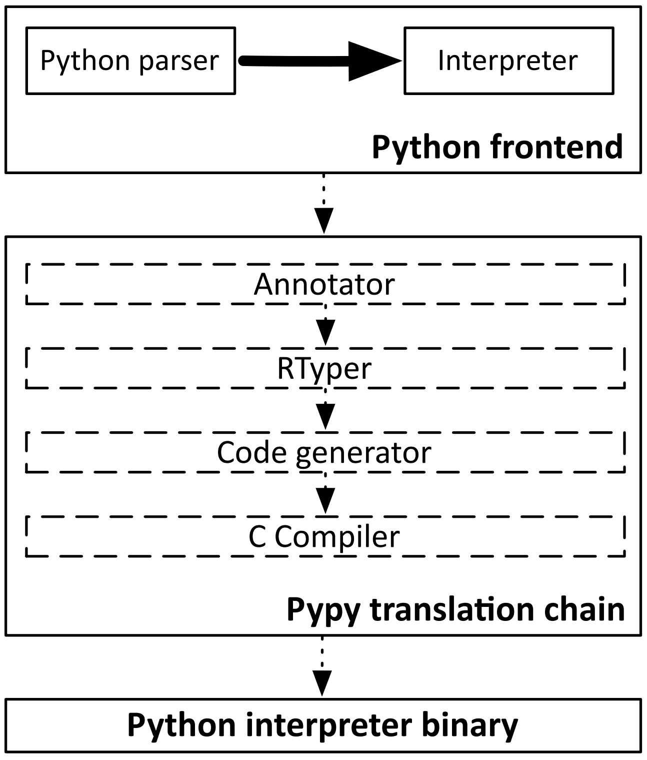


Figure 1. High-level structure of PyPy.

consists of a Python bytecode compiler and interpreter written in a restricted subset of Python called RPython [12]; these front end components are then passed through a source-to-source translator that generates the correspoding C code, then passed to a C compiler. The binary that results from this process is a full Python interpreter. Unlike Python, RPython is a statically-typed language. In C, the program developer manually specifies the types of all variables. However, in a RPython program, the types are inferred by an algorithm executed by the *Annotator* component of PyPy. Next, the RTyper converts these annotations from abstract types into concrete ones supported by the code generation back end, which are the types from the C language in our case. All auxiliary operations that operate on high-level RPython data structures, such as lists and dictionaries, are also converted to inline code or function calls at this step. Then, the source-to-source translator has been extended with its own JIT compiler, that traces the execution of the interpreter at run-time and converts the so-called "hot loops" (loops that have a large number of iterations) to optimized assembly code, using information gathered at run-time. Bolz et al. describe this JIT compiler in more detail in their paper [15].

Since PyPy has such a modular design, it is easy to write a new front end for a new language. Several such front ends exist for languages such as Javascript, Scheme and Prolog [6, 16]. Given all these prior successes, we have implemented a new interpreter front end in PyPy for the PHP language. Our project reuses all the existing compilation code from PyPy; the only new code components we had to implement are the PHP parser (which we did not implement from scratch, but reused the parser from Zend in a way which we will present later) and the interpreter loop, along with all data structures necessary for representing the data of a PHP program in memory.

3. Design and Implementation

Our prototype is a PHP interpreter written in RPython that is compiled into a binary by the same PyPy backend described before. Since any program written in RPython must be compilable to C, the types of its variables must be computable by the type inference algorithm in PyPy and supported by the low-level type system. The type of a variable is not computable if it is too general, i.e., a specific RPython type cannot be determined for that variable. For example, a single variable cannot be assigned an integer and later a string; the variable must either be always assigned an integer or always assigned a string. Since PyPy is currently under development, a complete specification of the RPython language is unavailable; however, most of its restrictions are available in the official PyPy documentation [5].

3.1 The PHP Bytecode Compiler

We use a design similar to the Python interpreter in PyPy. We convert PHP programs into bytecode. In order to reduce implementation effort, we reuse the PHP parser from the Zend engine. Zend also converts scripts internally to a linear bytecode before executing them. The engine does not provide direct access to this bytecode. Fortunately, several external extensions do export this bytecode to a format that can be accesses from an external application [1, 24].

The PHP bytecode consists of data structures called *oparrays*, which are arrays of all operations in one function along with other auxiliary information, such as lists of the named, temporary and static variables defined in the function. There is one outer *oparray* for the top level of a script, one array for each function and class method. The main element of the *oparray* is a pointer to a vector of `zend_op` structures, one for each operation in the function. Each structure contains the opcode and one `zend_znode` structure to hold the result and up to two operands. Most bytecode operations use only one or two operands; a few also use the extra `extended_value` field in the structure or two more operands encoded using the `ZEND_OP_DATA` opcode. The interpreter skips any operation with this opcode during interpretation.

The *Advanced PHP Cache*, or APC [1], is an accelerator for the Zend engine that stores values into persistent storage between executions. These values can be either key/value pairs that the user stores and reads manually or internal PHP data that the engine stores itself. The most significant example of the latter are the programs themselves in bytecode format. APC dumps the in-memory representation of the bytecode to a buffer or to disk. Our implementation reads this representation back into memory and interprets it.

The buffer generated by APC is a heavy-weight representation of all data structures used by Zend in memory. Some of the information stored therein is not used by our interpreter; other parts are represented in a non-linear way. Direct interpretation of this format is not the most efficient approach; to increase the efficiency of the interpreter, we convert from the Zend bytecode to our own bytecode, using the *BcParser* component shown in Figure 2. The *oparray* from Zend is not cache-friendly and requires many indirect memory accesses to get all the information for one operation. Thus, we convert each operation to a fixed-length (currently, 9 words each 32-bits long) representation that is contiguous in memory, as shown in Figure 3. The first word is the opcode, followed by 2-word encodings of the result and the operands, consisting of a type and integer value for each one. Each operand is either an immediate integer or an index into an array, in the case of non-integer constants or variables. The last two values are the field `extended_value` from the Zend structure and a jump hint value that we use to jump out of loops. We reuse the opcode numbers from Zend, including the `ZEND_OP_DATA` extension.

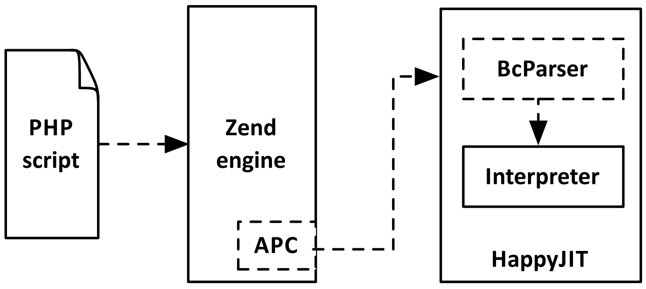


Figure 2. Passes that a PHP script goes through before being executed.

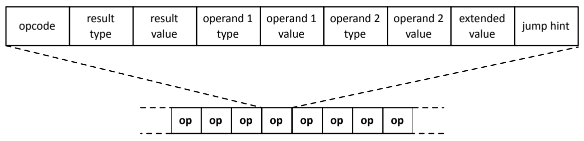


Figure 3. Memory layout of HappyJIT bytecode.

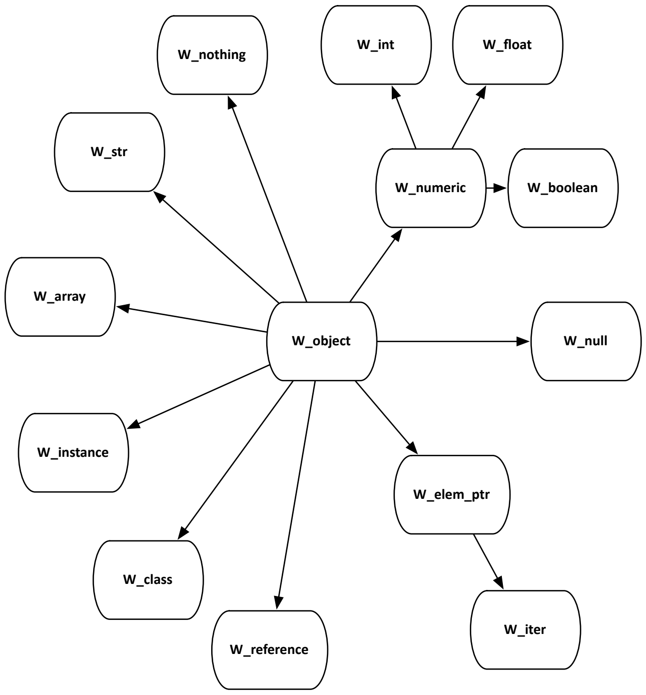


Figure 4. Hierarchy of types in HappyJIT.

3.2 The Basic Data Types

Variables in PHP have one of several types: *null, boolean, integer, float, string, array, object* and *resource*. We represent each constant or variable from a PHP script in RPython as an object that encapsulates the PHP value. There is one wrapper object type for each PHP type, except for *resource*, which we did not need to implement. These types are organized as a hierarchy, which can be seen in Figure 4. All the basic types in PHP other than *resource* have a corresponding HappyJIT wrapper object: *W_null, W_boolean, W_integer, W_float, W_str, W_array, W_instance*. The implementation stores two kinds of information for PHP objects: instance objects, one for each PHP instance, stored using *W_instance* structures, and class objects, one global object per class named *W_class*. All types are derived from a base class called *W_object*. We never construct objects with this type, so no instance of this type should appear. However, its presence makes it very easy to check whether a HappyJIT value is wrapped or not, simply by checking if that value is an instance of the base type.

We introduced the *W_nothing* wrapper as a placeholder for the `None` Python value, since RPython did not allow us to mix `None` with values of other types as the type of the same variable. This type of object is also used wherever there is an invalid or impossible value, such as uninitialized variables or the empty index used to append to arrays, like in the instruction `$a[] = 10`. It must be distinguished from the *W_object* root, since all types in our hierarchy descend from this root, so any `isinstance(obj, W_object)` check on a wrapped object will be true.

Most binary operators in PHP are either arithmetic or logical operators, which means the operands and result are numeric values. In some cases where the operands are not numeric, the values are coerced to a numeric type. In order to simplify the operations, we have put all three numeric types under one generic super-type, called *W_numeric*. Each object has a function `to_numeric` which either converts that object to one of the numeric types or returns the object itself if it already has the proper type.

The interpreter stores strings as instances of *W_str*. In order to be competitive with the Zend implementation, which uses NULL-terminated strings and offers random-access reads and writes to string elements in constant time and linear-time concatenation, we store strings not as RPython strings, but as lists of characters. Using RPython strings posed a major performance problem, since they are implemented as immutable objects. PHP allows the programmer to replace or take a reference to any character in a string. Implementing this using immutable strings would have made these stores take a time proportional to the length of the string. Using lists, the time per operation is constant: only one element in the list needs to be replaced. In Figure 5, the character *b* changes to *x*. Using immutable strings, the equivalent RPython operation would have been `a = a[0:1] + 'x' + a[2:]`. In our implementation, `a` is represented as `['a', 'b', 'c']`, which is a mutable data structure.

```
<?php
$a = 'abc';
$a[1] = 'x';
?>
```

Figure 5. PHP script that replaces one character in a string.

The interpreter creates objects of type *W_class* that contain the descriptions of PHP classes when the bytecode contains the `ZEND_FETCH_CLASS` operation. This operation triggers the lookup of a given class in the class table stored in the bytecode file. That class is then inserted in the global table of classes and a wrapper is created for it.

3.3 Arrays and Iterators

The implementation of arrays has been even more challenging, due to the flexibility that PHP arrays offer. In PHP, arrays are dictionaries indexed by either integer or string keys. They support not only random-access get and set operations, but the *append after last element* operation, where the last element of an array in PHP is defined as the one with maximum integer key. All these operations must also have constant time performance, in order to be competitive with the official PHP implementation. Since the data structure must support string keys, it must be implemented as a dictionary. However, we have noticed that significant performance can be gained when optimizing for dense arrays that have only integer keys; sparse arrays with integer keys would have far too great memory requirements, so they are also implemented as dictionaries. The Lua language also offers the same interface for arrays as PHP. In Lua 5.0[21], the virtual machine stores the values of the array that have integer keys from a small range in a separate linear array.

In our implementation, arrays are wrapped in the *W_array* structure, which stores a pointer to a back end object which represents the actual storage of the data. There are two kinds of back ends: *linear* back end, which stores the data as a linear array, and the *dictionary* back end, which stores everything in a dictionary. Each *W_array* object is initialized with a pointer to an empty *linear* back end. Whenever this back end must perform an operation that it does not support, it must *degenerate* to the other back end, which supports all operations. The *degeneration* operation constructs a new empty *dictionary* back end, copies all the data from the old one to the newly constructed one and then discards the old data structure. All following operations are executed on the new structure.

Although iterators are a PHP feature that is not directly visible to the programmer in PHP, Zend uses them internally to implement the `foreach` loop statement. This statement has one of two forms: `foreach ($array as $value) { ... }` or `foreach ($array as $key => $value) { ... }` and executes the statements in the brackets once for each element in the array. The second form also computes the key of each element. The value variable can also be specified as a reference, in which case it receives a reference to the original element instead of its value. The PHP bytecode compiler compiles this statement to a pair of bytecodes, `ZEND_FE_RESET` and `ZEND_FE_FETCH`. The former initializes an iterator at the beginning of the loop, while the latter advances the iterator by one position at the end of each iteration. There is one detail in the execution of these statements that makes their implementation more complicated: the iterator accesses a snapshot of the array taken at the beginning of the loop. Any changes to the array made inside the loop must not affect the iterator, especially when the items are not accessed by reference. The *W_iter* object stores an iterator over an array. It is initialized with a deep copy of the elements of the array, so that the array itself can be modified inside the loop.

Another relevant detail of the PHP language implementation is that assigning an array by the = operator will create a fresh copy, which can be subsequently modified without interfering with the left hand side. Passing arrays by value is also possible with the same copying semantics—if a function updates its array parameter, the update must be invisible for the array passed as actual parameter by the caller.

However, the naive implementation of this copying semantics is a huge and unnecessary burden when the copy is used read-only. For example, this is the case for the builtin `count` function, which returns the element count for an array. Building recursive data structures such as trees or graphs also requires an efficient way of passing read-only arrays, as a higher order tree is usually a brand-new array keeping lower-order trees as elements, passed in as read-only parameters.

We have implemented a copy-on-write mechanism in order to eliminate these problems. Every assignment increases a shared copy's counter. Any later update to an array with a counter greater than one triggers a deep copy. Releasing these copies at scope exit is cheap, because the counter updates are only necessary for the array directly accessible through scope variables. Nested arrays counters do not need updating as the Zend bytecode generator takes care of fetching any nested array into a scope variable before an update.

One minor optimization we have implemented in HappyJIT is the reuse of existing containers. Instead of allocating a new wrap-

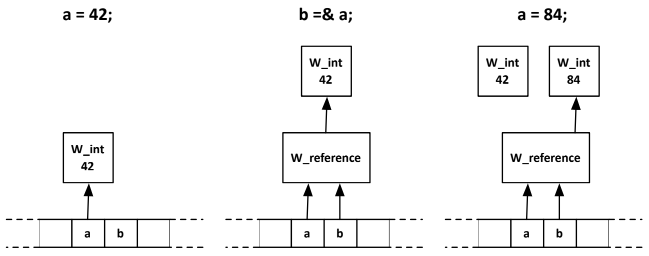


Figure 6. Change in storage locations after taking a reference to variable `a`.

per for the new value of a variable, HappyJIT puts an intermediate value in a temporary wrapper object, then stores that value directly into the target variable if the type matches. For example, when computing `$a = 2 + 3;` and `$a` is already an *W_int*, the interpreter does not store a newly allocated wrapper in the variable table. Instead, the new value, `5`, is stored directly into the existing object for `$a`. This prevents an unnecessary release of an expired object and allocation of a new one.

3.4 PHP References

Another major feature of the PHP type system are references. Using the `=&` reference-assignment operator, one variable `$a` can become a reference (alias) of another variable or storage location. These include array elements and object members (known as *properties* in PHP). Any stores to `$a` will be reflected on the original location as well. These references are effectively pointers to other variables; however, there is no PHP equivalent to the NULL pointer from C. All references always point to some live wrapped object.

Python does not provide any direct equivalent to these values, so we have integrated them into our wrapped object hierarchy. Whenever an operation from the script takes a reference to a location, the contents of that location are replaced with a *W_reference* object containing a pointer to its old data. The old wrapped data that was stored in the location still exists, but another level of indirection appears between it and its users. Whenever the interpreter accesses the referred data, it must go through the *W_reference* object. Figure 6 shows an example of this change. The variable `$a` is initially assigned the integer 42, so the variable table contains a pointer to a *W_int* that stores 42. Later variable `$b` gets a reference to `$a`, so both variable table cells are replaced with pointers to the same new reference object. Later, when the first variable is assigned the new value 84, the object is changed to point to the new wrapped value.

The *W_reference* objects keep track of aliases in the active scope in the `alias_count` member. Reference assignment when the right hand side is already a *W_reference* is implemented by shallow copying it and incrementing the `alias_count`. Naturally, if the left hand side is a *W_reference*, its `alias_count` is decremented before. As PyPy provides sophisticated garbage collection, reference counting for everything isn't needed—the scope of `alias_count` is limited to reference tracking. Zend uses the pair of `refcount` counter and `is_ref` flag for every entity as general purpose information for freeing unreachable objects, reference handling and copy-on-write mechanism. While possible in our implementation, this approach brings an unnecessary memory burden and a slight overhead for assignments. Instead, the more specific approach of counting only aliases (references) and the shared read-only copies described earlier, using independent counters, was chosen for both simplicity and efficiency.

An alternative approach would be to store everything as a reference right away. The downside of this seemingly orthogonal and elegant approach is that the additional indirection for every read and write incurs a performance penalty. Given the fact that refer-

```
$a = array(1,2,3,4,5);
foreach ($a as &$a[1]) { ... }
foreach ($a as $a[2] => $a[1]) { ... }
```

Figure 7. Example of `foreach` with and without references.

ences are a rather exotic feature of the PHP language, we had to take this path in order to optimize the more frequent use case.

There is a catch however, as we have not mentioned anything about what happens when all aliases but one fall out of any scope. The current implementation settles for keeping the "degenerate", single alias reference as it is. There is no incentive for going back to a plain value whenever the alias count reaches 1.

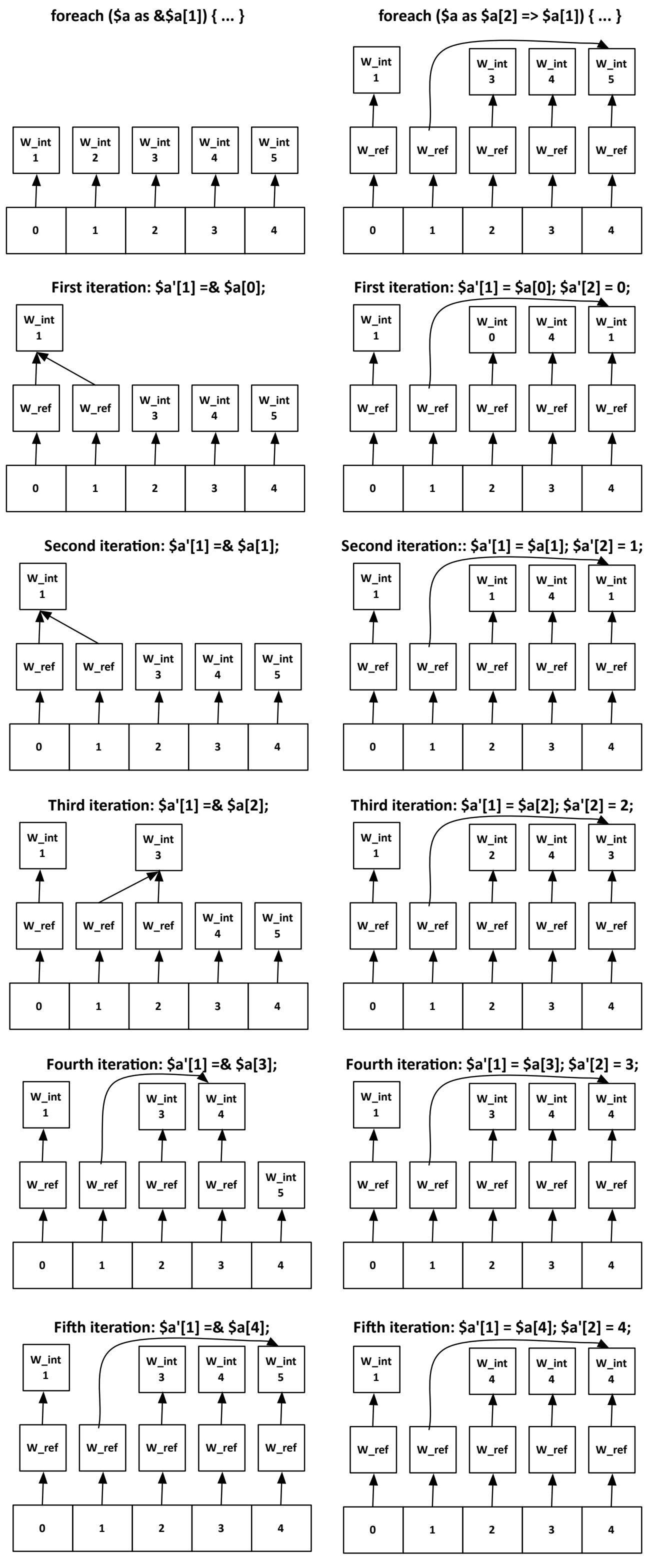
It is still necessary to transform a degenerate (single alias) reference to a plain value whenever such a reference is assigned to another location. We needed this approach because of the way we implemented references and the way the *foreach* language construct works. Let us take a look at the code snippet in Figure 7. In each iteration of the first loop, a reference to the current element is assigned to `$a[1]`. Because we are promoting both the left hand side and the right hand side, all elements of the array will contain references. However, at the end of the first loop, only `$a[1]` and `$a[4]` will have an alias count of 2. The other elements will subsequently lose the `$a[1]` alias during the next iteration. Therefore, the next time we are iterating through the array, the proper references (having an alias count more than 1) will be copied as references, as every update to them must be seen by the iterator as well. On the other hand, blindly copying degenerate references would be wrong, since the language semantics requires those being treated as values instead of references.

The second loop has a slightly more complicated behavior. Three observations must be made here: first, the bytecode compiler generates code that assigns the values of the key and array value using a regular PHP assignment, but in reverse order, i.e., the value is assigned first, then the key. Second, assigning a value or a reference to itself has no effect, whether the assignment is by reference or not. The last observation is that each loop operates on a copy of the array taken at the beginning of the loop; each key/value pair receives a value from this snapshot, not from the current array data.

We initially considered representing references as the indices of the referred locations in the variable table. This approach seems correct, at first glance. The problem is that a function variable (or an element of an array) can escape a function to its caller. If that function is later called again, a new value might overwrite the location that just escaped from the old call. Therefore, the value would need to be moved from the vanishing call frame to a different storage. This would add an additional level of indirection to the reference access, just like our current solution, so we abandoned this approach.

3.5 The Interpreter Loop

Our main implementation contribution is the interpreter loop. A short excerpt of this loop appears in Figure 9. The first few operations load the opcode and operands of the next operation from the linear bytecode array into loop variables. The rest of the loop is a series of `if` statements, one for each implemented operation. Each statement checks if the opcode matches. If so, the interpreter executes the implementation, then advances the program counter and continues execution at the beginning of the interpreter loop. This is effectively a RPython implementation of a `switch`-based interpreter instruction dispatch mechanism. There are various other dispatch mechanisms for interpreters that increase performance, but they provide no advantage when used in a traced interpreter [27].



foreach ($a as &$a[1]) { ... }
foreach ($a as $a[2] => $a[1]) { ... }
First iteration: $a'[1] =& $a[0];
First iteration: $a'[1] = $a[0]; $a'[2] = 0;
Second iteration: $a'[1] =& $a[1];
Second iteration:: $a'[1] = $a[1]; $a'[2] = 1;
Third iteration: $a'[1] =& $a[2];
Third iteration: $a'[1] = $a[2]; $a'[2] = 2;
Fourth iteration: $a'[1] =& $a[3];
Fourth iteration: $a'[1] = $a[3]; $a'[2] = 3;
Fifth iteration: $a'[1] =& $a[4];
Fifth iteration: $a'[1] = $a[4]; $a'[2] = 4;
W_int
W_ref

Figure 8. An array `$a` after two `foreach` loops. The `$a` array is the original array from the start of each loop, while `$a'` are the values visible inside the iteration.

```
while pc >= 0:
    ...
    opcode = bytecode[pc * 9]
    res_type = bytecode[pc * 9 + 1]
    res_val = bytecode[pc * 9 + 2]
    ...
    if opcode == bcp.ZEND_ASSIGN or
       opcode == bcp.ZEND_QM_ASSIGN:
        if op2_type != bcp.IS_UNUSED:
            runtime_ctx.assign_value(op1_val,
                        op2_type, op2_val)
        if res_type != bcp.IS_UNUSED:
            runtime_ctx.assign_value(res_val,
                        op1_type, op1_val)
        pc = pc + 1
    elif opcode == bcp.ZEND_ECHO or
         opcode == bcp.ZEND_PRINT:
        to_echo = runtime_ctx.get_wrapped_value(
                     op1_type, op1_val,
                     global_ctx.nullObject)
        print_str(to_echo.deref().to_str())
        pc = pc + 1
    ...
```

Figure 9. Excerpt of code from the intepreter loop.

Let us take a look at how the PyPy JIT compiler optimizes this code. We will use the term *application loop* to identify a hot loop in the PHP script, while the interpreter loop is the main loop described earlier. An iteration of an *application loop* corresponds to one or more iterations of the interpreter loop, since more than one bytecode operation might be executed by the script in that iteration. Because all *application loops* correspond to the same single interpreter loop, PyPy uses a system of "interpreter hints" that the interpreter gives to the JIT compiler so that the latter may differentiate between separate loops in the script. Each *application loop* is uniquely identified by a `(script, pc)` pair which identify the script that contains the loop and the position of the first instruction of the loop (the loop header). The PyPy JIT uses these tuples as keys in a hash table that contains all the currently known *application loops*, augmented with counters which keep track how many times each loop has been executed. The interpreter notifies the JIT whenever the PHP bytecode contains a backwards jump instruction, that is, to a previous position in the bytecode. The JIT then adds the destination of that jump to the list of loop headers and increments the execution counter for that loop. Whenever that counter reaches a threshold, the loop is considered hot and compiled to native code. Yermolovich et al. have used a similar design to implement a Lua interpreter written in ActionScript [29]. In their implementation, the ActionScript tracing JIT compiler would optimize Lua-level loops based on hints given by the Lua interpreter to the ActionScript VM.

The compiler then eliminates all the loads and opcode checks whenever a hot application loop gets compiled into native code. All opcodes that occur in the loop are fixed at the time of JIT-compilation, so the sequence of opcodes becomes an *application loop* constant. The compiler then performs constant folding, so all checks of the form `opcode == ZEND_XXX` are either evaluated to a true or false constant. For each interpreter loop iteration, the compiler eliminates all code but the one for that iteration's opcode, along with `if` check for that iteration. For this reason, we have not seen any justification to implement any other kind of interpreter, e.g., a threaded interpreter. In hot loops, the overhead of loading the operation and transferring control to the handler completely disappears.

The execution context of the interpreter is split into three parts: the *global context*, the *script context* and the *runtime context*. Our loop is designed so that more than one program can be executed before termination; it executes programs serially in the order specified at startup time. Since each program has its own bytecode and various other structures, we have grouped these into the *script context*. The *global context* stores a list of the scripts, along with a few other immutable global data, such as the table of builtin functions. These two context data structures are initialized when the interpreter is started and remain unchanged for the entire run. We have grouped the rest of the execution state, like the values of global and static variables for the current script, the function argument, local variable stacks, current function index and the program counter, into the *runtime context*. Since each execution of a program must start with clean copies of these structures, the *runtime context* is reset at the start of the script.

In any *oparray*, there are several different kinds of variables: global, static, local and internal. Using processor architecture terminology, the internal variables can be compared to an infinite set of virtual registers, while the other types of variables are comparable to the memory. All bytecode operations accept only internal variables as operands. The internal variables are further split into a few types: *temporaries*, *vars* and *compiled vars*. Earlier versions of the Zend engine loaded the values of all other kinds of variables into internals, did all the computation on these and then stored back the results in the global/static/local variables. *Temporaries* were used to compute the values of expressions, while *vars* are used to store values across operations. Recent versions of PHP have added the third kind as an optimization. Instead of being placed separately and loaded for each use, local variables are assigned one *compiled var* so they can be used directly as operands.

In our implementation, we do not distinguish between internal and other types of variables. We store all variables in one linear stack of wrapped objects. The *runtime context* contains this array. Each *oparray* contains the number of total internal variables, so we preallocate the space for those whenever a function is called. All the named local variables are simply pushed on the stack whenever the program first references each one. We also use the linear variable stack to store the call frame for each function. When a function calls another function, the interpreter stores some of the caller state, that is popped back later on return: the caller's index in the function table, the program counter of the call operation, the bottom and top of the caller's stack frame and the old function's count of parameters.

Before a function calls another function, the caller pushes the callee's parameters on the stack. The bytecode provides a few opcodes just for this purpose, all starting with `ZEND_SEND`. However, since these parameter operations can be intertwined with other operations that might push values on the stack, we could not use the same stack for variables and parameters. Otherwise, some sequence of operations might push a parameter, then a variable, then another parameter. The callee would be unable to access the parameters as a contiguous region on the stack and pop them on return. There is a global stack for parameters stored in the *runtime context*, which callers push arguments on, while the callees pop off the appropriate number of values on return.

A similar mechanism exists for function names. Whenever the script contains a function call, the interpreter must do the following operations: identify the function by its name, compute and push the arguments on the argument stack, then perform the actual function call. Each of these operations corresponds to some bytecode operations that the bytecode compiler generates. These steps must be done in this exact order, as the interpreter must know

```
<?php
$s = 0;
for ($i = 0; $i < 10000; $i++) {
    $s += $i;
}
?>
```

Figure 10. Example of PHP script with loop.

which function is called before executing any ZEND_SEND operation. This is crucial for error checking, as errors such as parameters sent by the caller as values and expected by the callee as references must be reported. The Zend engine uses a fast path for this case where the function is known at parse-time. This occurs when it is a regular function call (as opposed to an object method) and its name is a string constant, known at parse-time. In this case, the bytecode compiler simply does not emit anything for the first case, but encodes all information about expected arguments in the ZEND_SEND operations. In any other case, the bytecode compiler generates a ZEND_INIT_FCALL_BY_NAME operation before any of the other parameter-related operations. This operation must lookup the function's name in the global function table and compute its index in that table. All the following ZEND_SEND operations and the function call itself use this index to identify the function and its format parameters. Therefore, this index must be stored in memory in a location accessible by these operations. For this purpose, we use a third stack, the *function name stack*. ZEND_INIT_FCALL_BY_NAME pushes the index on the stack, then ZEND_DO_FCALL_BY_NAME pops it back when performing the call.

There were two data structures we considered for the stack of variables: a linked list of frame objects, one per function, which would include the storage for all the local variables in that function, or a single linear array of stack locations that all functions push and pop values from. The Python interpreter in PyPy is an example of the former approach, as it allocates on PyFrame object per function, while programs writted in C and compiled to X86 code are a good example of the latter. In programs with recursive or frequently called functions, allocating one frame object per call leads to a lot of redundant memory allocations, where frames are repeatedly allocated and released after a short time. To prevent this, we have implemented the frames using the latter approach, stored on a global linear stack. This stack is preallocated as an array of *W_int*s that is "large enough" for most scripts and extended whenever needed. This also has an impact on the performance of the code generated by the JIT compiler, as it is able to fold some constant pointers to locations on the stack and avoid other unnecessary object constructions and destructions. For example, if the same function is called many times, the five values in its frame need not be allocated as wrapped objects, but simply stored as integers in the existing slots.

3.6 Loop Example

Let us take a look at the intermediate and final representations that a PHP program goes through. Figure 10 shows a small PHP program with a loop. Figure 11 contains the bytecode generated by the compiler for this program. Operands prefixed with ! correspond to compiled variables. In this case, !0 is the variable $s while !1 is the loop variable $i. The operands starting with the ~ are temporaries which do not correspond to any variable in the source code. Statements 7 and 8 compose the main body of the loop, where the addition in the loop takes place.

The bytecode generated by this loop is interpreted by HappyJIT using the C implementation of the interpreter. When a sufficient number of iterations have been executed, the RPython implementa-

line	#	op	ext	return	operands
2	1	ASSIGN			!0, 0
3	2	ASSIGN			!1, 0
	3	IS_SMALLER		~2	!1, 10000
	4	JMPZNZ	8		~2, ->10
	5	POST_INC		~3	!1
	6	FREE			~3
	7	JMP			->3
4	8	ASSIGN_ADD	0		!0, !1
5	9	JMP			->5
7	10	ECHO			'%0A'
	11	RETURN			1

Figure 11. Corresponding Zend bytecode for script from Figure 10.

```
# Loop 0 : loop with 45 ops
[i0, p1, p2, ..., i22, i23]
debug_merge_point('Pc:5', 0)
p24 = getarrayitem_gc(p4, i5, descr=...)
guard_class(p24, 9074280, descr=<Guard2>)
p26 = getarrayitem_gc(p4, i8, descr=...)
guard_nonnull_class(p26, 9074280, descr=<Guard3>)
i28 = getfield_gc(p24, descr=...)
setfield_gc(p26, i28, descr=...)
i29 = getfield_gc(p24, descr=...)
i31 = int_add(i29, 1)
debug_merge_point('Pc:6', 0)
debug_merge_point('Pc:7', 0)
debug_merge_point('Pc:3', 0)
p32 = getarrayitem_gc(p4, i9, descr=...)
setfield_gc(p24, i31, descr=...)
guard_nonnull_class(p32, 9074280, descr=<Guard4>)
i34 = getfield_gc(p32, descr=...)
i35 = int_sub(i34, i11)
i37 = int_lt(i35, 0)
guard_true(i37, descr=<Guard5>)
p39 = getarrayitem_gc(p4, i12, descr=...)
guard_nonnull_class(p39, 9075208, descr=<Guard6>)
debug_merge_point('Pc:4', 0)
p41 = getarrayitem_gc(p4, i13, descr=...)
setfield_gc(p39, 1, descr=...)
guard_class(p41, 9075208, descr=<Guard7>)
i44 = getfield_gc(p41, descr=...)
i45 = int_is_zero(i44)
guard_false(i45, descr=<Guard8>)
debug_merge_point('Pc:8', 0)
p46 = getarrayitem_gc(p4, i17, descr=...)
guard_class(p46, 9074280, descr=<Guard9>)
p48 = getarrayitem_gc(p4, i22, descr=...)
guard_class(p48, 9074280, descr=<Guard10>)
i50 = getfield_gc(p46, descr=...)
i51 = getfield_gc(p48, descr=...)
i52 = int_add(i50, i51)
p53 = getarrayitem_gc(p4, i17, descr=...)
guard_nonnull_class(p53, 9074280, descr=<Guard11>)
p55 = getarrayitem_gc(p4, i23, descr=...)
setfield_gc(p53, i52, descr=...)
guard_nonnull_class(p55, 9074280, descr=<Guard12>)
debug_merge_point('Pc:9', 0)
debug_merge_point('Pc:5', 0)
setfield_gc(p55, i52, descr=...)
jump(i0, p1, ..., i22, i23, descr=<Loop0>)
```

Figure 12. JIT loop generated from code in Figure 10.

tion of the interpreter switches into tracing mode and generates an efficient representation of the loop that is then compiled to binary code. Figure 12 shows this internal representation. The loop is simply a sequence of small operations that end in a jump to the start of the loop, with various guards inserted in different places inside the loop. The JIT compiler transforms each loop operation into a small number of assembly instructions, taking a very small amount of time to execute.

Before entering the loop, many values and addresses are precomputed and passed to the loop as constants specified in the list on the first line, e.g., the address of the local variable table as value `p4` and the number of iterations as `i11`. Each `debug_merge_point` pseudo-operation marks the start of an iteration of the interpreter loop. In some cases, bytecode operations are reduced to no-ops after loop generation. An example of this are control flow operations, since the loop is a linear piece of code given that all the guards succeed.

This loop also provides a good example of container reuse. A straightforward implementation of the `ASSIGN_ADD` opcode would add the values of `$s` and `$i` together and allocate a new *W_int* object for the result, then store this object into the variable table at the location corresponding to `$s`. Instead, the interpreter stores the primitive integer value directly into the wrapper that already contains `$s`. The sum of the two values is computed in the loop and saved in `p52` temporary, then stored at the memory location from `p53`. This location stores `$s`.

4. Evaluation

4.1 Performance Results

We tested our interpreter on a 64-bit 2.4GHz Intel Core i3-370M running Debian Linux, kernel version 3.0.0. We evaluated the performance of our interpreter when compared with the official PHP implementation, stable version 5.33.6. Two versions of our interpreter appear if the evaluation: one translated into C using PyPy (*Happy*) and the other one translated into a JIT compiler (*HappyJIT*). The results are the average of 10 runs. Our implementation was compiled with PyPy built from the repository, revision *29823:ce505f035396*.

This evaluation uses two test suites. The first one we have used is `PHPbench` [4], a well-known suite of benchmarks written specifically for the PHP language. The second test suite is a subset of the PHP tests from the *Computer Language Benchmarks Game* [8].

Figure 13 shows the relative performance of the tests in `PHPbench`. We have measured the execution times relative to the official PHP interpreter and the Roadsend PHP compiler [7]. Most tests prove to be faster on the JIT-compiler interpreter than on Zend or Roadsend, while the non-JIT version is slower than Zend. The slowdown of the latter is as high as 10x for one particular test, `variable_variables`, and 5x for some of the others. We couldn't get performance data on all tests for Roadsend, as it failed the `arithmetic` test from `PHPbench` and didn't finish the `nsieve-2` test from the `Shootout` suite.

There are a few tests that have significantly worse performance on our interpreter than on the standard one. The first poorly-performing test is `local_hash_assign`, which tests the performance of assignments to hash tables that use string keys. Our implementation uses the default implementation of dictionaries in RPython for this case. These data structures are implemented as C or low-level RPython code which does not get analyzed by the JIT compiler. All helper functions that these structures use are called by the JIT without being traced, so they cannot be optimized at run-time. The dictionaries are implemented as hash tables with open addressing, so each dictionary operation contains a loop with a number of iterations that depends on the key being inserted. This

File	Slowdown	Slowdown (no JIT)
binarytrees-3	4.18x	4.19x
local_array_assign	8.25x	5.73x

Figure 14. Slowdown from disabling copy-on-write.

number of iterations not only varies with the hash value of the key, but it is also not large enough to trigger the tracing of the loop. The JIT compiler could not optimize these operations anyway. The next test that shows a slowdown, `variable_variables`, uses variables whose names are only known at run-time. Therefore, any access to such a variable requires a dictionary lookup. This test shows a significant slowdown for this reason.

Our implementation of strings also poses some performance problems. The remaining three tests that have worse performance on our JIT all use the string concatenation operation provided by the PHP language. In our implementation, this operation is a simple concatenation of the lists of characters in the two strings, using the Python function `extend`. The Zend engine uses the C function `strcat`, which has been highly optimized by the writers of the C library and some compilers optimizing compilers also replace it with a very efficient built-in version.

Figure 15 presents the performance results for the *Computer Language Benchmarks Game* tests. The only tests where our implementation falls behind are `fibo`, `ackermann` and `binarytrees`. These tests are very similar in one respect: the piece of code most executed is a recursive function. Although our pre-allocated stacks and wrapper-reuse optimizations are designed to make function calls as fast as possible, recursive functions still prove to be slower under the JIT compiler than in Zend. Tests that contain one or more long-running loops, such as `nested loop`, `primes` and `nsieve`, show a significant performance improvement when JIT-compiled.

We have also analyzed the impact of the copy-on-write mechanism for arrays. Most of the tests we have run do not have any array copies that would benefit from this, so we observed no change in performance. Figure 14 shows the slowdown, relative to the default setting (copy-on-write enabled). Two tests, `binarytrees-3` from the `Shootout` suite and `local_array_assign` from `PHPbench` are heavily penalized when disabling this mechanism. This is to be expected, as both make heavy use of array assignment without changing the copies.

5. Related Work

Tracing JITs have gained enormous popularity in recent years. The origin of this concept is a binary rewriting system called Dynamo [13], where tracing was used to optimize pre-compiled binary code. This system would detect hot loops, convert them into traces, optimize the traces and generate new, more efficient binary code. The original version only supported linear traces; later, more complex data structures for traces appeared, such as trace trees [18]. Advanced tracing JITs using the more advanced data structures have been written for a variety of statically and dynamically-typed languages, such as the YETI [30] and HotPath[19] for Java, TraceMonkey [20] for JavaScript and TamarinTracing [17] for ActionScript. The latter was also used successfully as the lower layer of a layered virtual machine for the Lua language [29].

Several JITs have been built for the PHP language. One such implementation [23] used the JIT compiler in LLVM [22] to generate optimized code at run-time. It compiled the *oparray* operation handlers into LLVM bitcode offline, then convert the PHP bytecode into a linear sequence of LLVM calls to these handlers when the program was run. Using a combination of inlining and other LLVM optimizations, it would convert the bytecode into optimized native code, then directly execute that code. Another project, P9 [28], re-

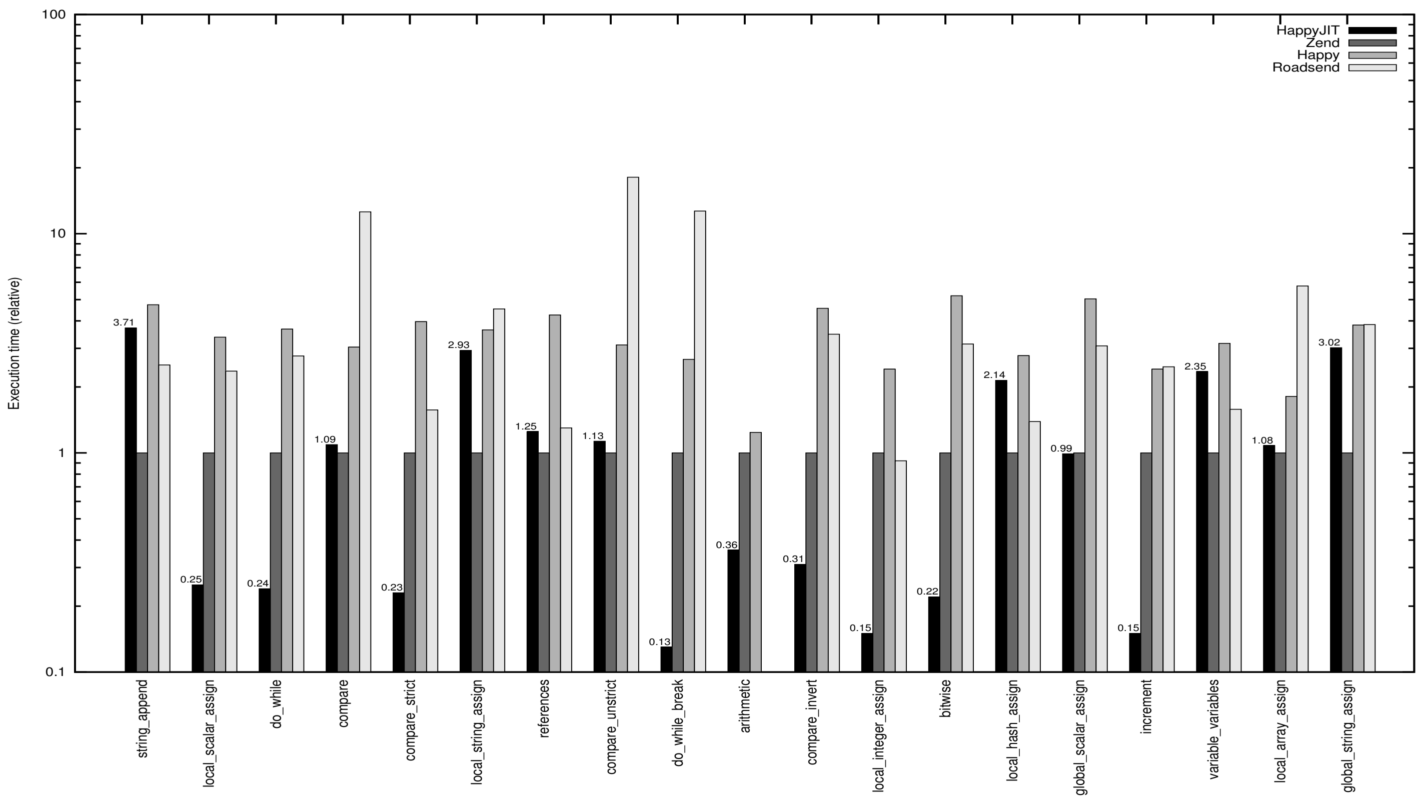


Figure 13. PHPbench tests performance, relative to Zend.

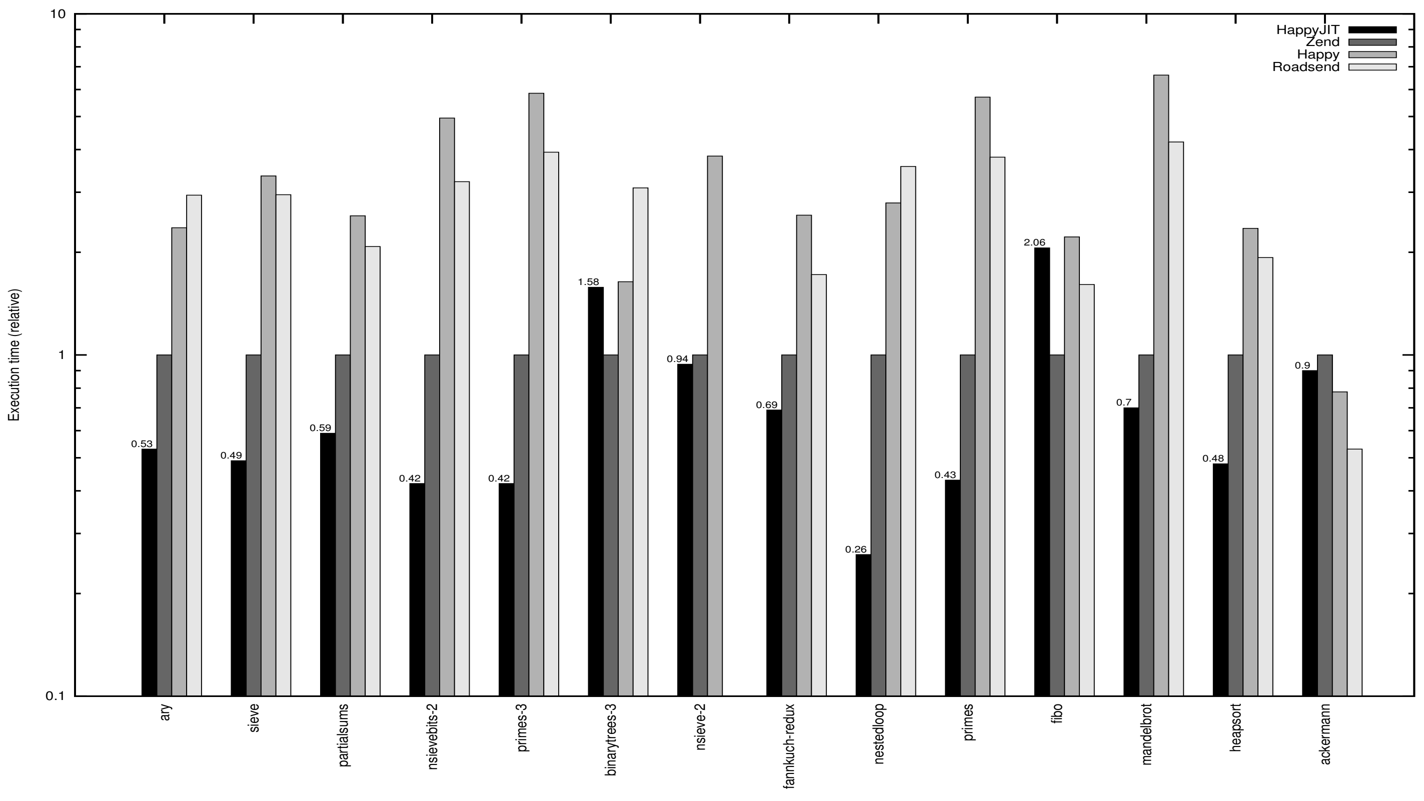


Figure 15. Shootout tests performance, relative to Zend.

targeted an existing JIT called TR JIT to a PHP front end. This JIT is also used in the IBM J9 Java VM. The authors adapted this JIT to the dynamic features of PHP and showed that there is a performance gain in just-in-time compilation for PHP.

Another approach for dynamically-typed language optimization is source-to-source translation. Instead of being compiled to some bytecode and then interpreted, the code written in the dynamically-typed language is converted to another language then compiled with an existing compiler. The target language is often a static language, such as C or C++. This idea is also used to as the first version of a compiler for a new language; for example, the first C++ compiler, *cfront* [2], converted the input C++ code to C. Several source-to-source translators for PHP have been written, such as *Hiphop* [3], *phc* [14] and *Roadsend PHP* [7]. The first one converts a given PHP program into C++ code, attempting to keep the object-oriented structure of the program intact after translation. PHP classes are converted to C++ classes and PHP functions are converted to C++ functions, wherever possible. The second compiler, *phc*, only targets the C language, simulating object-oriented programming using the features offered by C. The third compiler converts a program into a mix of C and Bigloo Scheme [26]. As PHP is a dynamically-typed language, there are some parts of the program code that are not completely known until run-time, e.g., dynamic variable names or calls to the `eval` function. Such code cannot be precompiled, so these source-to-source translators either must also implement a complete PHP interpreter or not support some dynamic features of the language. Moreover, some static analysis for type inferrence [11] must be done in order to avoid at least some of the wrapping and late binding overhead. More often than not this leads to very poor results because of a combinatorial explosion of possible types inherent to a dynamically-typed language. JIT compilation of PHP bytecode does not have this drawbacks and can optimize statically and dynamically-typed code equally, as all the optimizations are done at run-time.

6. Conclusions and Future Work

We presented the design of an interpreter for the PHP language written in RPython, along with many details of the data model used by this interpreter. The design of this system was heavily motivated by the need to support all the features of the PHP language that are not directly visible to the programmer, such as iterators, in an efficient way. We have also shown that running our interpreter using a tracing JIT compiler provides significant performance improvements for some common use cases. JIT compilation has proven to be a very effective optimization technique for the PHP language.

We have identified a few weak spots in the performance of our compiler. There is a performance penalty for function calls which could matter for highly recursive code. We plan to investigate whether this can be solved with more careful pre-allocation of frames and higher level, language aware optimizations.

In addition, a frequent scenario that appears in the execution of PHP programs is the repeated interpretation of the same program, many times. If the program itself does not contain a loop, a tracing JIT will not compile the program to native code. We are looking to adapt the interpreter loop so that "hot programs" are identified, traced and JIT-compiled. We have already experimented with tricking PyPy into handling entire programs the same way as loop bodies. The results are good, trivial programs without loops showing the same speedup as if their code was inside a loop. However, some work on PyPy itself is needed in order to properly handle alternating between different programs. As PyPy is a meta-JIT, using very few assumptions about what loops are, we are confident that this can be done without any disruptive changes in the PyPy codebase. Most of the work for recording and retrieving traces for non-consecutive loop iterations is already done. Moreover, the `loop_longevity` JIT parameter controls the lifespan of inactive (currently not used) traces before being discarded—controlling the trade-off between speed and memory usage should be therefore possible out of the box.

We have already mentioned in the introduction one major use of PHP programs. In this paper, we have evaluated the performance of our prototype interpreter on synthetic benchmarks. It would also be interesting to evaluate the performance of HappyJIT in more realistic scenarios, such as the server-side page generators presented earlier. To this end, we must either integrate our interpreter into an existing webserver such as Apache or build a new server from scratch around our program. This will enable us to evaluate statistics like page generation time and memory usage and compare them against the statistics of other implementations. At the time of writing this paper, neither our PHP language support nor our PHP runtime were complete or stable enough to allow for such tests.

Acknowledgments

We would like to thank Stefan Brunthaler for the great feedback and advice on this paper and Michael Franz for his guidance and feedback. We are deeply grateful to the anonymous reviewers for the valuable comments and suggestions that helped us improve this paper.

References

[1] The Alternative PHP Cache. `http://pecl.php.net/package/APC`.

[2] cfront. `http://www.softwarepreservation.org/projects/c_plus_plus/cfront`.

[3] HipHop for PHP. `https://github.com/facebook/hiphop-php`.

[4] The PHP Benchmark. `http://www.phpbench.com/`.

[5] PyPy official documentation, . `http://codespeak.net/pypy/dist/pypy/doc`.

[6] PyPy language front ends, . `http://codespeak.net/svn/pypy/lang/`.

[7] Roadsend PHP. `http://www.roadsend.com/home/index.php`.

[8] The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.

[9] Unladen Swallow. `http://code.google.com/p/unladen-swallow/`.

[10] The Zend PHP engine. `http://www.zend.com/en/community/php/`.

[11] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 1995 European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60160-0.

[12] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Dynamic Languages Symposium*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8.

[13] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

[14] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 Symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8.

[15] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 2009 Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3.

[16] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for Prolog execution. In *Proceedings of the 2010 International Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 99–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0132-9.

[17] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 International Conference on Virtual Execution Environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4.

[18] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006.

[19] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2006 International Conference on Virtual Execution Environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8.

[20] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[21] R. Ierusalimschy, L. H. D. Figueiredo, and W. Celes. The Implementation of Lua 5.0. *Journal of Universal Computer Science*, 11: 2005.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 0:75, 2004.

[23] N. Lopes. Building a JIT compiler for PHP in 2 days, 2008.

[24] D. Rethans. The Vulcan Logic Dumper. `http://derickrethans.nl/projects.html`.

[25] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for Python. In *Proceedings of the 2004 Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 15–26, New York, NY, USA, 2004. ACM. ISBN 1-58113-835-0.

[26] M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Proceedings of the 1995 International Symposium on Static Analysis*, SAS '95, pages 366–381, London, UK, 1995. Springer-Verlag. ISBN 3-540-60360-3.

[27] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4:2:1–2:36, January 2008. ISSN 1544-3566.

[28] M. Tatsubori, A. Tozawa, T. Suzumura, S. Trent, and T. Onodera. Evaluation of a just-in-time compiler retrofitted for PHP. In *Proceedings of the 2010 International Conference on Virtual Execution Environments*, VEE '10, pages 121–132, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-910-7.

[29] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 2009 Symposium on Dynamic languages*, DLS '09, pages 79–88, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1.

[30] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a gradually extensible trace interpreter. In *Proceedings of the 2007 International Conference on Virtual Execution Environments*, VEE '07, pages 83–93, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1.