Static Optimization in PHP 7

Nikita Popov   Biagio Cosenza   Ben Juurlink
Technische Universität Berlin, Germany
nikic@php.net, {cosenza, b.juurlink}@tu-berlin.de

Dmitry Stogov
Zend Technologies, Russia
dmitry@zend.com

Abstract

PHP is a dynamically typed programming language commonly used for the server-side implementation of web applications. Approachability and ease of deployment have made PHP one of the most widely used scripting languages for the web, powering important web applications such as WordPress, Wikipedia, and Facebook. PHP's highly dynamic nature, while providing useful language features, also makes it hard to optimize statically.

This paper reports on the implementation of purely static bytecode optimizations for PHP 7, the last major version of PHP. We discuss the challenge of integrating classical compiler optimizations, which have been developed in the context of statically-typed languages, into a programming language that is dynamically and weakly typed, and supports a plethora of dynamic language features. Based on a careful analysis of language semantics, we adapt static single assignment (SSA) form for use in PHP. Combined with type inference, this allows type-based specialization of instructions, as well as the application of various classical SSA-enabled compiler optimizations such as constant propagation or dead code elimination.

We evaluate the impact of the proposed static optimizations on a wide collection of programs, including micro-benchmarks, libraries and web frameworks. Despite the dynamic nature of PHP, our approach achieves an average speedup of 50% on micro-benchmarks, 13% on computationally intensive libraries, as well as 1.1% (MediaWiki) and 3.5% (WordPress) on web applications.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.3.4 [*Programming Languages*]: Processors—Optimization

Keywords PHP, static optimization, SSA form

1. Introduction

In order to keep pace with the rapidly increasing growth of the Web, web application development predominantly favors the use of scripting languages, whose increased productivity due to dynamic typing and an interactive development workflow is valued over the better performance of compiled languages.

PHP is one the most popular [1] scripting languages used for the server-side implementation of web applications. It powers some of the largest websites such as Facebook, Wikipedia and Yahoo, but also countless small websites like personal blogs.

CC'17, February 5–6, 2017, Austin, TX, USA
© 2017 ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033026

In order to support its more dynamic features, PHP, like many other scripting languages, has traditionally been implemented using an interpreter. While this provides a relatively simple and portable implementation, interpretation is notoriously slower than the execution of native code. For this reason, an increasingly common avenue to improving the performance of dynamic languages is the implementation of just-in-time (JIT) compilers [2], such as the HHVM compiler for PHP [3]. On the other hand, JIT compilers carry a large cost in terms of implementation complexity.

In this work, we pursue a different approach: purely static, transparent, bytecode-level optimization. By this we mean that a) runtime feedback is not used in any form, b) no modification to the virtual machine or other runtime components is required and c) optimizations occur on the bytecode of the reference PHP implementation. The latter point implies that, unlike many alternative PHP implementations, we must support the full scope of the language, including little used and hard to optimize features.

This static approach is motivated by the PHP execution model, which uses multiple processes to serve short-running requests based on a common shared memory bytecode cache. As this makes runtime bytecode updates problematic, many dynamic optimization methods become inapplicable or less efficient. We pursue interpretative optimizations partly due to the success of PHP 7, whose optimized interpreter implementation performs within 20% of the HHVM JIT compiler for many typical web applications [4].

Our optimization infrastructure is based on static single assignment (SSA) form [5] and makes use of type inference, both to enable type-based instruction specialization and to support a range of classical SSA-based optimizations. Because PHP is dynamically typed and supports many dynamic language features such as scope introspection, the application of classical data-flow optimizations, which have been developed in the context of statically typed languages, is challenging. This requires a careful analysis of problematic language semantics and some adaptations to SSA form and the used optimization algorithms.

Parts of the described optimization infrastructure will be part of PHP 7.1. Our main contributions are:

1. A new approach to introducing SSA form into the PHP language, including adaptation for special assignment semantics and enhancement of type inference using π-nodes.
2. The implementation and analysis of a wide range of SSA-enabled optimizations for a dynamic language.
3. An experimental evaluation on a collection of micro-benchmarks, libraries and applications, including WordPress and MediaWiki.

The remainder of the paper is structured as follows: section 2 describes related work on dynamic language optimization. Section 3 presents relevant PHP language semantics and section 4 discusses the use of SSA form in PHP. SSA-enabled static optimizations investigated in this work are described in section 5. An experimental evaluation on micro-benchmarks, libraries and applications is presented in section 6. The paper closes with a discussion and conclusion in sections 7 and 8.

2. Related Work

SSA Static single assignment form [5] has become the preferred intermediate representation for program analysis and optimizing code transformations, and is used by many modern optimizing compilers [6–8]. Data-flow algorithms are often simpler to implement, more precise and more performant when implemented on SSA form. Typical examples include sparse conditional constant propagation [9] and global value numbering [10]. More recently, SSA form has become of interest for compiler backends as well, because the chordality of the SSA inference graph simplifies register allocation [11].

Specific applications often require or benefit from extensions of the basic SSA paradigm. Array SSA form [12] modifies SSA to capture precise element-level data-flow information for arrays for use in parallelization. Hashed SSA form [13] extends SSA to handle aliasing, by introducing additional μ (may-use) and χ (may-define) nodes. The ABCD algorithm [14] introduces π-nodes to improve the accuracy of value range inference. In this work, we further extend this idea for use in type inference.

A focus of recent research has been on the formal verification of SSA-based optimizations [15–17], as well as SSA construction [18], and destruction [19].

Dynamic language optimization Many different approaches to improving the performance of traditionally interpreted dynamic languages have been investigated. The most successful in terms of raw performance are JIT compilers [2].

Another avenue is the translation of code to a lower-level language. For example, the Starkiller project [20] translates Python code to C++, using an augmented Cartesian product algorithm [21] for type inference. However, this approach is often not able to support all language semantics.

Run-time feedback can be integrated into interpreters in a number of ways. Dynamic interpretation [22] interprets a flow graph that models not only control flow but also type uncertainty. Würthinger et al. [23] use an abstract syntax tree (AST) based interpreter on the premise that modification of ASTs to incorporate runtime feedback is simpler than modification of bytecode. Brunthaler [24] approaches the problem of dynamic bytecode updates by adding an additional inline cache pointer to each instruction.

The overhead of the virtual machine itself may also be reduced. Threaded code [25], superinstructions and replication [26] reduce indirect branch misses. Favorable instruction scheduling reduces instruction cache misses [27].

PHP optimization The wide adoption of the PHP language has motivated the development of several projects aiming at improving its performance.

The undoubtedly most significant one is the HipHop Virtual Machine (HHVM) [3] developed by Facebook. HHVM uses a JIT compiler operating on tracelets, which are regions of code with a single entry but potentially multiple exits. Tracelets are symbolically executed in a single-pass, forward data-flow analysis annotating instructions with input and output types, where statically unknown input types are observed at runtime. Type guards at the start of the tracelet allow optimization to proceed using mostly complete type information. If a type guard fails, the tracelet can be compiled with another set of input types.

The precursor of HHVM is the HipHop compiler (HPHPc) [28], which compiles PHP code to C++. The compiler specializes the generated code based on types inferred using an adaptation of the Damas-Milner constraint-based algorithm [29]. No bytecode representation is used, instead all operations are performed on the AST level. HPHPc does not support some of PHP's dynamic language features and requires all code to be known in advance.

The phc compiler [30] also translates PHP to C. A large focus of the phc implementation is on accurately modeling the aliasing behavior of references. To achieve this, flow- and context-sensitive alias analysis, type inference and constant propagation are performed simultaneously and prior to construction of Hashed SSA form. In our work we will largely ignore this aspect, because accurate handling of references has become much less important after PHP 5.4 removed support for call-time pass-by-reference. Additionally, issues that will be discussed in section 3.5 effectively prevent this kind of analysis if PHP's error handling model is fully supported.

A number of alternative PHP implementations leverage existing JIT implementations. Phalanger [31] and its successor Peachpie [32] target the .NET CLR, while Quercus [33] and JPHP [34] target the JVM. HippyVM [35] uses the RPython toolchain. While many of these projects report improvements over PHP 5, they cannot achieve the same level of performance as a special-purpose JIT compiler such as HHVM.

3. Optimization Constraints

PHP supports a number of language features that complicate static analysis. In the following, we discuss how they affect optimization and also justify why we consider certain optimization approaches to be presently impractical. Some of the mentioned issues apply to many scripting languages (dynamic typing), while others are PHP specific (references). As we operate on the bytecode of the reference PHP implementation, a few implementation-specific constraints are also covered.

While the following discussion primarily deals with features that inhibit optimization, there are also two properties of the PHP language that make it more amenable to static optimization than many other scripting languages: First, PHP has a strictly separated function scope and requires global variables to be imported explicitly. Second, PHP does not support runtime replacement of functions or methods ("monkey-patching").

3.1 Dynamic and Weak Typing

PHP is a dynamically typed language, which means that types of variables are generally only determined at runtime and may vary. Additionally the type system is weak, by which we mean that use of mismatched types in operations generally does not lead to an error, and is instead handled through implicit and potentially lossy type conversions. For example `"foo"*"bar"` evaluates to integer zero, because the non-numeric strings are cast to zero prior to multiplication.

Additionally, it is common for basic operations to return different result types depending on the types and values of their operands. For example, the addition operator may have an integer, a floating point number, an array, or an overloaded object as the result type. This result type overloading makes it harder to statically infer types.

3.2 References

With the exception of objects and resources, PHP uses by-value argument passing and assignment semantics by default. For example, if an array is passed to a function, any modifications to it will not be visible outside the function. References provide a mechanism to circumvent this by creating a mutable cell which can be shared by multiple variables. Figure 1 shows a basic usage example. References may be created dynamically and conditionally, so that it cannot always be statically determined whether or not a variable holds a reference. A comprehensive analysis of references and their interaction with copies can be found in [36].

```
$var1 = 42;
$var2 = &$var1;
$var2 = 24;
var_dump($var1); // int(24)
```

Figure 1. Basic references example. `$var1` and `$var2` end up pointing to a shared mutable storage location.

Particularly problematic for optimization is the ability to specify functions as accepting arguments by-reference, as shown in Figure 2, because there is no indication that by-reference argument passing is used at the call-site (only at the declaration-site). This implies that we have to pessimistically assume by-reference passing if we cannot determine the callee statically.

```
function inc(&$n) { $n++; }
$i = 1;
inc($i);
var_dump($i); // int(2)
```

Figure 2. By-reference argument passing. Variable `$i` is converted into a reference during the call.

3.3 The Use-Def Nature of Assignments

In many languages an assignment to a variable has no behavioral dependence on the previous value held by this variable: an assignment constitutes a definition of the variable, but not a use. This does not hold in PHP for a number of reasons.

Firstly, assignments to references can be understood in terms of a pointer write `*ptr = val` in C. In this case the variable `ptr` itself is only read, while the write occurs to the location it points to. The same is true for references in PHP. Secondly, if the assigned-to variable is the last holder of an object with a destructor, the assignment will execute it.[1] This requires knowing the previous value of the variable, constituting a use. Lastly, the assignment operator may be overloaded, in which case it behaves similarly to a method call, again constituting a use.

Due to these cases, we have to assume that any assignment acts as both a use and definition of a variable. As will be discussed later, this has a significant impact on the structure of the SSA graph and commonly requires special treatment during optimization.

3.4 Dynamic Scope Modification

PHP supports dynamic scope introspection through "variable variables," as shown in Figure 3. In this example, `$varName` stores the name of a different local variable `$var` and then indirectly modifies it using the `${$varName}` syntax.

```
$var = 42;
$varName = 'var';
${$varName} = 24; // behaves as: $var = 24;
var_dump($var);   // int(24)
```

Figure 3. Variable variables example. `$var` is modified indirectly using its name stored in `$varName`.

There are a number of other ways to perform similar operations. For example, the `extract()` function allows the extraction of an associative array into the local scope. A particular concern for optimization is that such a function might be called dynamically, as shown in Figure 4. This would severely limit optimization, because various special handlers (e.g., autoloading and error handlers) can perform implicit function calls in many situations. To circumvent this problem, we have submitted a language change proposal to forbid dynamic calls to scope introspection functions, which has been accepted for PHP 7.1 [37].

```
$i = 42;
$fn = 'extract';
$fn(['i' => 24]); // behaves as: $i = 24;
var_dump($i);     // int(24)
```

Figure 4. The `extract()` function, which extracts an associative array into the local scope, is called dynamically. This is forbidden as of PHP 7.1.

This allows us to detect all dynamic scope introspection statically, in which case we exclude the function from optimization entirely. The reason for this choice is pragmatic: dynamic scope introspection is extremely rare in real applications. As such, a more fine-grained approach, such as treating variable-variable assignments as potential modification points for all variables, is not worthwhile.

3.5 Error Handling

Next to exceptions, the primary error handling mechanism in PHP are runtime warnings. Unlike exceptions, these warnings do not abort the current execution path. Instead, they are displayed or logged, and an error-indicating value such as *null* is returned from the offending operation. The possibility of such an error-indicating value reduces the quality of type inference results.

The more significant problem, is the possibility of registering an error handler, which is invoked whenever a runtime warning is triggered. As nearly all operations in PHP have error conditions, this means that arbitrary code can run at nearly any point in a function. This makes application of optimizations to global variables (which may be modified by the error handler) infeasible.

Even worse, the variable scope in which the error occurred is passed as an argument to the error handler. While this generally does not allow modification of variables, it *does* allow arbitrary changes to references, as well as object properties. This possibility effectively prevents us from performing type analysis on references or object properties, even if they are otherwise local to the function. (This is a good example of how a single ill-considered feature can significantly limit optimization work.)

3.6 Pseudo-main Scope

A PHP file can, next to declarations for functions, classes, etc., also contain freestanding code, referred to as pseudo-main code. Such code will adopt the scope from the location it was included in (for a top-level include this would be the global scope).

Figure 5 illustrates why this kind of scope-adoption impedes optimization: through clever use of an object destructor, it is possible to change the result of a simple addition, even though all variables were explicitly assigned beforehand. Together with the possibility of performing modifications through an error handler, this makes the pseudo-main scope highly unpredictable. As such, we exclude it from optimization. This is not problematic for modern PHP code, which is (apart from some initialization code) fully contained in classes or functions, but it does limit applicability to legacy code.

3.7 Type Annotations

PHP supports annotating function signatures with argument and return value types. Unlike similar features in some other scripting

[1] While PHP makes no strict guarantees, it is generally understood that (non-circular) object destruction occurs as soon as possible.

```
// file1.php
$a = 1;
$b = 1;
var_dump($a + $b); // int(2) via file1.php
                   // int(5) via file2.php

// file2.php
$b = new class {
  function __destruct() {
    $GLOBALS['b'] = 4;
  }
};
include 'file1.php';
```

Figure 5. Example of scope-adoption in pseudo-main scope. The behavior of `file1.php` is significantly altered if included through `file2.php`, which forces a specially crafted scope.

languages, these type annotations do not merely serve static analysis, but are actually enforced at runtime. However, types are only checked at call boundaries, so that an argument `int $n` will ensure that `$n` is an integer on entry into the function, but will still allow a subsequent assignment of a different type, such as `$n = "str"`. Nonetheless, these type annotations provide valuable type roots for use in type inference.

However, type annotations for scalar types such as booleans, integers and floats (which are the most useful for inference) have only been introduced in PHP 7 and as such are not yet in wide use (and consequently played no role in our experimental evaluation). We expect that these type annotations will become more useful for optimization in the future. Additionally, it is likely that type annotation support will be expanded to include object properties at some point [38]. This would be especially valuable, because it circumvents the issue discussed in section 3.5, which prevents inference of object property types.

3.8 Execution Model and Virtual Machine

PHP applications are commonly deployed based on a shared-nothing architecture, where each incoming request is handled starting from a clean slate. By default this also applies to the compiler: all used scripts have to be tokenized, parsed and compiled to bytecode (called *opcodes* in PHP) anew on each request. Because this carries significant overhead, all performance-sensitive deployments additionally make use of an opcode cache (*opcache*), which caches the compiled bytecode for files in shared memory (SHM). As compilation time is less important in this configuration, opcache also contains the optimization infrastructure which we are extending.

PHP applications are not compiled as a whole. Instead, individual files are included at runtime, at which point they are either compiled or loaded from SHM. It is possible to reference symbols that will only be defined at a later point in time without forward declarations. Additionally, if opcache is used, all files are compiled completely independently (without knowledge of previously defined symbols) to avoid cache dependencies. This limitation of the current architecture is significant, because it implies that we do not know the signature of any function defined outside the current file, and must pessimistically assume all arguments to be passed by reference, as discussed in section 3.2.

As our goal is to perform purely static and transparent bytecode optimizations, we have to work within the framework of the current PHP virtual machine (VM). The instruction format is essentially a three-address code with at most two input and one output operands, though sometimes input operands are modified in-place. The VM supports two main kinds of variables: compiled variables (CVs) correspond to actual variables in the program code (such as `$foo`), while temporary variables are introduced by the compiler to hold intermediary results.

Both variable kinds have very different lifetime semantics: Compiled variables are initialized when a function is entered and destroyed when it is left. Instructions referencing CVs do not consume the variables, as such it is possible to use the same variable in multiple instructions. Conversely, temporary variables are not initialized upfront, instead the compiler ensures that they are only read after an explicit assignment. If an instruction uses a temporary variable, it is also responsible for destroying its value. Consequently, temporary variables can only be read once.

In both cases, there exists a strong coupling between value lifetime and storage location, which is one of the main factors distinguishing a VM variable from a CPU register. For example, this means that simply copying one variable to another will generally not preserve program semantics due to changes in value lifetimes.

4. SSA Form in PHP

Most of the analysis passes and optimizations described in the following operate on SSA form, whose defining property is that each variable is assigned at most once, while ϕ-nodes are used to merge values at control flow join points. We make heavy use of SSA form, because it greatly simplifies the implementation of flow-sensitive data-flow analysis passes, such as type inference, and allows to directly associate data-flow properties, such as inferred types, with variables, without loss of accuracy.

We implement SSA construction based on the classic algorithm due to Cytron et al. [5], which places ϕ-nodes based on iterated dominance frontiers. The necessary dominator tree is constructed using the simple data-flow algorithm due to Cooper et al. [39]. To reduce the number of unnecessary ϕ-nodes we employ pruned SSA form: iterative liveness analysis is used to determine live variables for all basic blocks, and ϕ-nodes are only placed for variables that are live-in at the respective block.

To ensure that the SSA form is strict, which means that every variable use is dominated by its definition, we place an implicit $v_0 = \mathrm{undef}$ assignment in the entry block for every compiled variable v. Use of such a variable (i.e., use prior to initialization) is allowed in PHP, but triggers a runtime warning.

Due to the issues described in section 3.3, assignments in PHP have to be treated as both use and definition points of the assigned variable. While this poses no fundamental problem to the SSA paradigm, it does significantly change the structure of the SSA graph and commonly requires special handling in analysis passes. Figure 6 shows the control flow graph (CFG) for a simple `while`-loop with a block-local variable v, using ordinary SSA form (left), and using SSA form where assignments are treated as both use and definition points (right). The notation $(v_1 \rightarrow v_2) = \ldots$ signifies that the assignment uses the old value v_1 and generates the new value v_2. Because the value of v_1 does not matter in most cases, we call this an *improper use*.

Because our goal is to perform transparent bytecode optimizations, it is convenient to match the existing bytecode format as closely as possible when translating into SSA form. For this reason we consider the SSA graph as an overlay structure: each bytecode instruction is associated with an "SSA instruction" specifying which SSA variables the three instruction operands use and define. If an operand both uses and defines a variable, this constitutes an in-place modification, such as $(v_1 \rightarrow v_2) \mathrel{+}= 1$ (while the notation is similar, v_1 is a proper use here).

One appeal of SSA form is that it makes it easy to implement control-flow-sensitive data-flow algorithms. However, there are cases where the variable splitting performed by SSA construction is not sufficient to capture some path-sensitive properties. An

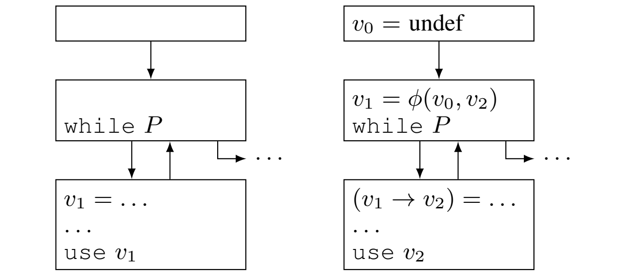

Figure 6. SSA form representation of simple `while`-loop with block-local variable v. Left: Ordinary SSA form. Right: Assignments are treated as both use and definition points, introducing an improper v_1 use and requiring the placement of an additional ϕ-node.

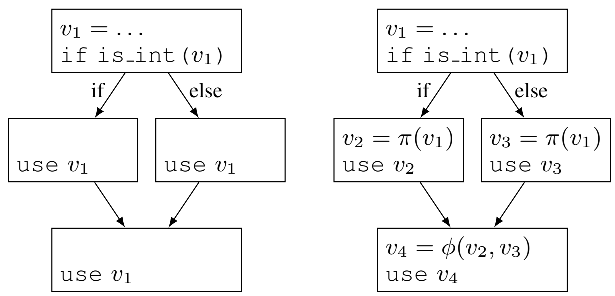

Figure 7. Left: Example CFG where control-sensitive type information cannot be captured. Right: π-nodes are introduced to artificially split SSA variables, so that code paths with control-sensitive type information use distinct names.

example is shown in Figure 7 (left), where v_1 must (or cannot) be an integer only on certain code paths. Because no distinct variable name is associated with these paths, this information is lost if types are directly associated with variables, rather than (variable, program point) pairs, as is usual for SSA algorithms.

This problem is solved by artificially splitting variables using π-nodes, as is shown in Figure 7 (right). A π-node is placed at the start of both branches, thus creating separate variables v_2 and v_3, with which the more precise type information can be associated. The concept of π-nodes is adopted from the ABCD bounds check elimination algorithm [14], where π-nodes were used to improve the accuracy of value range inference, rather than type inference.

5. Static Optimization

Based on the bytecode in SSA form, it is now possible to implement various analysis and optimization passes, which will be described in the following. The main supporting analysis is type inference, which is used for type specialization and plays a supporting role in constant propagation, dead code elimination and copy propagation. Finally, inlining is applied to increase the applicability of other optimizations.

5.1 Type Inference

Nearly all optimizations discussed in the following depend, in one way or another, on the availability of type information for variables. As PHP is a dynamically typed language, type information is not available a priori and instead needs to be inferred. To this purpose we make use of a generalized variant of the Sparse Conditional Constant Propagation (SCCP) algorithm [9].

If we abstract SCCP away from the specific application of constant propagation, the algorithm may be briefly summarized as follows. Each SSA variable is associated with an element from a bounded lattice $(L, \sqsubseteq, \top)$. The variables are optimistically initialized to the $\top$ value and a monotonic transfer function is evaluated for each instruction, which combines the lattice values of input operands to produce new lattice values for output operands (for ϕ-nodes this is the lattice meet). If this changes the value of a variable, all instructions using it need to be reevaluated. Using this procedure, lattice values are successively lowered until a fixed point is reached. At the same time, the algorithm keeps track of which CFG edges are executable given the current lattice state and only blocks (and ϕ-operands) that are currently executable will be considered. Once again, the starting point is an optimistic assumption that everything but the entry block is not executable. Together this results in an algorithm that is sparse and optimistic, and combines data-flow propagation with detection of unreachable code, making it more powerful than either on their own.

For type inference in particular the lattice may be approximated as a power set lattice $(\mathcal{P}(T), \supseteq, \emptyset)$ over a type universe T. Each element of the lattice is a set of types $S \subseteq T$, representing the possible types a variable might take at runtime. PHP supports eight fundamental types, namely *null*, *bool*, *int*, *double*, *string*, *array*, *object* and *resource*, where *bool* may be further subdivided into the pseudo-types *true* and *false*. For arrays we additionally track the possible key types (only *int* and *string*), as well as the possible value types, to one level of nesting. For objects we optionally store a specific class/interface type, while distinguishing whether this is the exact type of the object, or subtypes are allowed as well.

In addition to this proper type information, we also track whether a variable may be undefined (*undef*) or a reference (*ref*). *ref* also implies a union of all other types (*any*), as we do not track the type of reference variables (section 3.5). Variables are initialized to $\emptyset$, apart from the implicit variables in the entry block, which are *undef*. This lattice allows an accurate description of the possible types of a variable, with some limitations. In particular, nested arrays may not be represented accurately and it is not possible to represent unions or intersections of object types.

For this lattice the meet operation is given by the set union, while using the lowest common unique ancestor for objects that specify a specific type. The transfer functions for non-ϕ nodes model the (often very complicated) rules for the output types an instruction may produce given certain input types. The current type information is also used to determine whether CFG edges are executable. Common cases where this is applicable are type-checks of the form `is_int($v)` or `$v instanceof A`. However, while this may eliminate unreachable branches, this by itself does not make full use of the conditional type information: it does not capture that inside a branch guarded by `is_int($v)`, the variable `$v` will be an integer. To make use of this fact, we use π-nodes with associated type constraints as described in section 4, such that the variable type is intersected with the type constraint associated with the π-node.

Type narrowing The type inference algorithm as described, is a pure forward propagation algorithm: it starts from known type information primarily in the form of literal initializations and propagates this information forwards through the SSA graph. This is to be expected, as we are not allowed to infer additional type constraints on value sources such as parameters. However, there is one particular case where modifying the source of a type is both possible and desirable: While PHP distinguishes between integers and doubles, programmers commonly initialize variables using integers (`0` instead of `0.0`), even if they will only be used as doubles subsequently. This results in unnecessary *int*|*double* unions. To avoid this, after the main type inference pass has finished, we promote integer initializations to use doubles if this both eliminates such a type union and we can determine that the promotion does not change observable results (e.g., through loss of precision).

Value range inference In PHP, if the result of an integer arithmetic operation exceeds the integer range, it is promoted to double. As such, value range inference is necessary to accurately infer types on integer operations. We use the intra-procedural portion of [40] for range inference, which may be briefly summarized as follows. An interval lattice $(\mathcal{Z} \times \mathcal{Z}, \sqsupseteq_i)$ with $\mathcal{Z} = \mathbb{Z} \cup \{\pm\infty\}$ is used, where $\pm\infty$ denote under- and overflow and $\sqsupseteq_i$ is a super-interval relation. For a fixed number of warmup passes, intervals are updated based on instruction-specific transfer functions, as in the type inference algorithm. For variables that have not reached a fixed point, bounds are then widened to $\pm\infty$ depending on whether the variable is increasing, decreasing or both. In a final step, these conservative bounds are narrowed again, based on π-node constraints. These constraints may also depend on other variables (futures). Widening and narrowing occurs as per [41]. The entire procedure is not performed on the whole SSA graph at once, but on its strongly connected components (computed excluding improper uses) in topological order. This is important both for the efficiency and precision of the algorithm.

5.2 Constant Propagation

For constant propagation, the SCCP algorithm in its original form is used, with the lattice elements given by $\bot \sqsubseteq C_i \sqsubseteq \top$, where $\top$ represents an underdefined value (not yet known, may be constant), the C_i represent specific constant values and $\bot$ represents an overdefined value (not constant). In this case, the important property of the lattice meet is that $C_i \sqcap C_j = \bot$ for $i \neq j$, such that two distinct constants combine into an overdefined value. Variables are optimistically initialized to $\top$, unless they are implicit (undefined) definitions in the entry block, in which case $\bot$ is used instead.[2]

SCCP can be directly applied to PHP with only a few additional considerations: Firstly, we need to ensure that values of (potential) reference variables are not propagated, as these could change at any time (within the limits of our model). In most cases this will be automatically handled correctly, because any instruction that may produce a reference will produce a $\bot$ value during constant propagation. Only assignments of the form $(v_1 \to v_2) = w$ require explicit handling: if v_1 is $\bot$ *and* type inference has marked it as a potential reference, v_2 should be set to $\bot$ as well.

Secondly, PHP has a relatively broad concept of compile-time constants, that also includes strings and arrays. This is problematic, because propagation through chains of string or array operations may degenerate to quadratic space and time complexity, as each operation needs to copy the result of the previous one. This can be avoided in general by imposing size restrictions, but for specific common cases, copies may be avoided altogether by exploiting the fact that, for linear strands in the SSA graph, only the final value is significant.

Lastly, because type inference and constant propagation are based on the same underlying algorithm, it is easy to run both in parallel by operating on a product lattice. This not only avoids an ordering problem, but also allows detecting a larger class of unreachable code than any order or repetition of the individual algorithms.

5.3 Dead Code Elimination

Dead code elimination (DCE) on SSA form is performed using a simple worklist algorithm. A set of root instructions is marked as live and this property is propagated backwards in the SSA graph: if an instruction is live, then any instruction that generates one of its operations must also be live. The liveness roots are given by instructions that may have side-effects, as well as all branch instructions. There exists a variant of this algorithm [5], which uses control dependence to also allow elimination of dead branches. We do not use this variant, as it requires the computation of reverse dominance frontiers, while only eliminating little additional code.

A major obstacle to performing DCE in PHP is that approximately 95% of all VM instructions have an error condition that may result in a runtime warning or exception being thrown. In many cases these error conditions are obscure edge-cases, but nonetheless they need to be considered as side-effects for the purpose of DCE. To reduce the number of liveness roots introduced in this manner, we use the inferred type information to check whether an error may be triggered for a particular combination of input types.

Some additional problems arise when considering simple assignments of the form $(v_1 \to v_2) = w$. Apart from the obvious side-effect if v_1 is a reference, eliminating such an assignment may cause a subtle change to destructor semantics. If v_1 might have a destructor, eliminating this assignment could delay its execution. Conversely, if w might have a destructor, eliminating the assignment could cause it to run earlier. In both cases, the assignment cannot be removed.

A further problem is posed by the fact that v_1 constitutes an improper use. As such, we do not consider it as a use for the purposes of DCE, i.e., we do not mark the instruction generating v_1 as live only because the assignment is live. While this approach is acceptable if v_1 is generated by an ordinary instruction, it also implies that ϕ-nodes whose result is only used improperly, will be considered dead on termination of the algorithm. This is not correct and removing these ϕ-nodes would violate SSA properties.

To avoid this, the actual elimination of dead instructions is performed in two phases. First, we remove all dead non-ϕ instructions. Then, all ϕ-nodes that are still used improperly are marked as live and this information is propagated backwards to the ϕ-sources. Only after this step can dead ϕ-nodes be removed.

5.4 Copy Propagation on Conventional SSA

Copy propagation eliminates copy operations of the form $v = w$ by replacing all uses of v with uses of w. On unrestricted SSA form performing copy propagation is very simple, because each variable is defined exactly once, so we do not have to account for the possibility of other assignments to v or w. However, performing copy propagation in this manner breaks *conventionality* of the SSA form, by which we mean that related SSA variables are no longer necessarily interference-free (have disjoint live-ranges). *Related variables* here refers to the transitive reflexive closure over variables that occur as source or target in the same ϕ-node, or are used and defined by the same operand of an instruction. This partitions the SSA variables into equivalence classes.

The important property of conventional SSA form is that translation out of SSA can be performed simply by dropping all variable subscripts and ϕ-nodes. Otherwise, the use of an out-of-SSA translation algorithm is required, which resolves interferences within one equivalence class. We initially considered using the SSA destruction algorithm by Boissinot et al. [42] for this purpose, but found that its application to a scripting language is problematic, primarily because precise control over value lifetimes is lost. This is not only a concern with regards to observable destructor behavior, but can also negatively affect performance: inserting additional variable copies can result in copy-on-write separation of large data structures, sometimes causing very large slowdowns. The fundamental underlying problem is the strong coupling between value lifetimes and storage locations.

[2] Undefined variables evaluate to null upon use, and could be propagated as such. However, this would require special treatment of the "undefined variable" runtime warning.

For this reason, we restrict copy propagation to cases that maintain conventionality. In particular, for an assignment $(v_1 \to v_2) = w_1$ we require that the variable v_2 not be live-out at any modification point of w_1 (whereby we mean any use of w_1 that defines a new w_i on the same operand) or live-in at any block that contains a ϕ-node using w_1. To additionally preserve the direct correspondence between equivalence classes and non-SSA variables, we further require that v_2 is not used in ϕ-nodes (unless their result is only used improperly) and that there are no in-place modifications of v_2, such as $(v_2 \to v_3) \mathrel{+}= 1$ (again excluding improper uses).

The necessary liveness checks are performed using the fast SSA liveness oracle due to Boissinot et al. [43]. Unlike many classical liveness algorithms, which provide sets of variables that are live at certain program points, this SSA-based algorithm only answers queries of the form "is variable v live at program point p?" In broad terms, the algorithm works by precomputing node sets depending only on the CFG, and using them to efficiently check whether a path from p to a use of v exists, which does not leave the subgraph dominated by the definition of v. A primary appeal of this approach is that the precomputed information may be invalidated only by changes to the CFG, but not the SSA graph.

5.5 Type Specialization

Many instructions of the PHP virtual machine need to implement different behavior depending on the type of the operands. Additionally, they need to handle a number of unlikely conditions such as undefined or referenced variables. This is somewhat mitigated through use of fast-path/slow-path splitting, such that the most common cases are handled using a minimal number of checks, before falling back to a generic implementation. Even so, a simple operation like the addition of two integers still has to perform two type checks, as well as an overflow check.

To avoid this overhead, we can specialize generic instructions to type-specific ones based on the inferred type information.[3] For the `ADD` instruction, one may introduce `ADD_INT` and `ADD_DOUBLE` operations, which only accept integer/double operands. If value range inference determines that the result cannot overflow, one can further specialize `ADD_INT` to `ADD_INT_NO_OVERFLOW`.

However, this kind of specialization is limited in scope, because it mostly targets basic arithmetic operations (where type-checks have large relative overhead) and additionally requires very precise type information (no type unions). A broader class of specializations is obtained by considering higher-level properties, such as whether a variable may hold a reference-counted value or not. In particular the types *null*, *bool*, *int* and *double* never use reference counting.

In this context, it is important to note that an operation like `$c = $a + $b` is compiled into a sequence of two instructions. First `T = ADD $a, $b` will write the result of the addition into a temporary `T`, and then `ASSIGN $c, T` will copy this result into the compiled variable `$c`. Non-temporary assignment is a separate instruction, because it involves complex logic in the general case (destroying the previous value safely, handling reference assignments, handling overloaded assignment operators, etc.) However, if type inference determines that `$c` cannot be reference-counted, both instructions may be combined into `$c = ADD $a, $b`. Similarly compound assignments like `ASSIGN_ADD $a, $b` may be converted into `$a = ADD $a, $b`, avoiding various checks related to in-place modifications.

Other examples of specialization include: Baking object property offsets into property fetch instructions, if the object type and property are known. Specializing argument sends for the common case of known defined, non-reference variables. Removing instructions entirely in some cases, e.g., for casts or type assertions.

5.6 Function Inlining

To reduce function call overhead and improve the applicability of other static optimizations, we implement a basic function inlining pass. Inlining occurs prior to SSA construction to avoid the need of keeping the SSA form representation of more than one function at the same time.

When inlining a function call, we do not only have to incorporate the instructions of the called function, but also its variables. For compiled variables this poses an issue, because such variables stay alive until the end of a function. To avoid changing program semantics because of this, we have to insert `UNSET_VAR` instructions after the inlined function body. This also ensures that variables are in a consistent state on (re)entry into the inlined function body.

While DCE will commonly be able to remove these additional instructions, it is not always the case. Additionally, all compiled variables must be initialized on entry into a function and destroyed on exit. For this reason inlining in PHP comes with additional overhead beyond the usual increase in program size.

Inlining in the framework of the current PHP implementation only has limited applicability: Due to the single-file restriction discussed in section 3.8, we can only inline functions defined in the same file. Additionally, for method calls the target is usually only exactly known for private and final methods. Inlining for virtual methods would require speculative devirtualization [44]. Of course, inlining cannot be used if either the inlined or inlined-to function uses overly dynamic language features such as dynamic scope introspection. Otherwise, the scope of both functions would potentially be visible.

For the experimental evaluation we used an aggressive inlining heuristic, which prefers to inline all eligible functions that are not excessively large (less than 500 VM instructions), to one level.

6. Experimental Evaluation

To evaluate the effectiveness of the optimizations described in the previous section, we use two sets of micro-benchmarks provided by the PHP distribution, as well as a small selection of real application and library code. In all cases, we consider execution time averaged over many runs and normalized against the baseline execution time. The baseline is provided by running the tests with SSA-based optimizations and inlining disabled, but other preexisting optimizations enabled.

The time to compile and optimize the code is not considered as part of the execution time. This is representative of practical usage, because the optimizer is part of the opcache extension and as such only used if opcode caching is enabled. In this case the compilation time is amortized across many requests.

The tests were performed on an Intel Core i5-2500K CPU with 8GB RAM running Ubuntu 16.04. For the tests that require a database backend, MySQL 5.7.13 was used. A web server was not used.

6.1 Micro-benchmarks

The PHP distribution comes with two sets of standard micro-benchmarks. The first (bench.php) implements a number of functions that either perform simple algorithms (e.g., computation of Mandelbrot sets or prime numbers) or certain code pattern (e.g., accessing arrays in specific orders). The results for these benchmarks are shown in Figure 8, with the geometric mean speedup being 1.26× without inlining and 1.50× with inlining.

We can make a number of observations about these results: Inlining does not affect most benchmarks, because they only use

[3] Because PHP VM instructions include a pointer to the instruction handler, this can be done without modifying the VM itself. It is not necessary to introduce dedicated opcodes, though it may be more convenient.

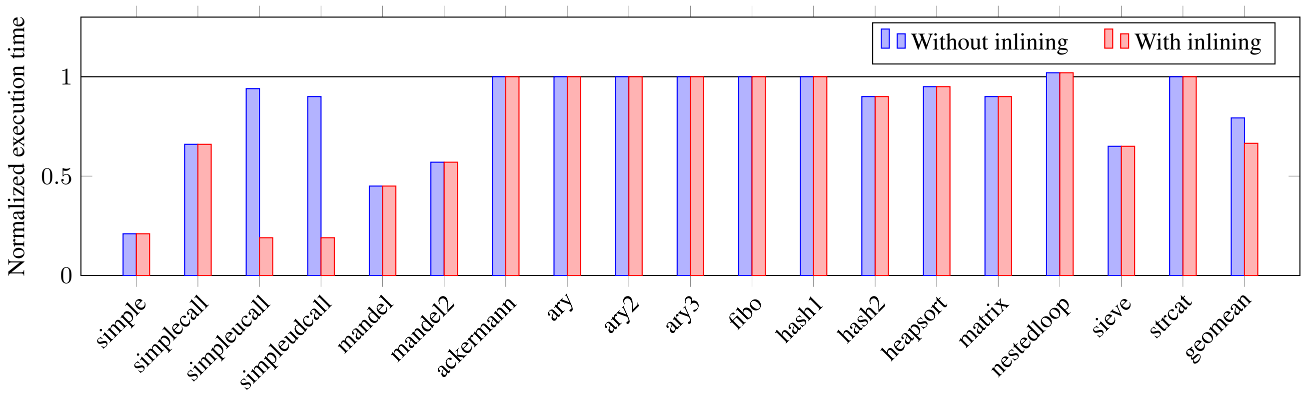

Figure 8. Normalized execution times for standard PHP micro-benchmarks for static optimization without inlining (blue bars) and with inlining (red bars). Baseline represented as black line, lower is better. The last column is the geometric mean.

a single function. The largest improvements (5.3×) are realized for simpleu(d)call, where inlining enhances DCE. For the remaining benchmarks, we observe between 2.2× improvements (mandel) to no change (ary1-3), depending on what type of operations are prevalent in the benchmark. Arithmetic optimizes well, while array manipulation does not.

For the benchmarks where our optimization strategy had non-zero impact, Figure 9 shows a breakdown illustrating how much the individual optimizations contributed to the speedup. These results have been obtained by measuring with only a single optimization pass enabled. However, to avoid complicating the interpretation with too much pass interdependence, inlining was always enabled.

The individual contributions do not always sum to one, because some passes enable others (e.g., transformations performed by assignment specialization can support further type specialization), while others feature some degree of overlap in their effects (e.g., copy propagation and assignment specialization sometimes have similar effects on bytecode).

The second set of micro-benchmarks distributed with PHP (micro_bench.php) are different in nature: they measure the repeated execution of a single operation, or a combination of very few operations. As such, these benchmarks provide little value, as they essentially only test whether we can successfully DCE a particular operation. DCE eliminates the loop body in 10 out of 34 cases. In the remaining cases we fail to prove that the operation can never generate a runtime warning.

6.2 Applications and Libraries

As performance improvements on micro-benchmarks commonly do not translate to realistic workloads, we additionally evaluate performance using a number of real-world PHP applications and libraries:

1. **Wordpress:** A popular blogging platform. Response time for many sequential executions[4] of the WordPress homepage, as populated by Facebook's oss-performance tool, is tested. (Version 4.2)
2. **MediaWiki:** The software powering Wikipedia. Response time for many sequential executions of the Wikipedia page of Barack Obama, as populated by oss-performance, is tested. (Version 1.26.2)
3. **phpseclib:** Library implementing pure-PHP fallbacks for cryptographic primitives. Execution time of encryption and decryption using RSA with 1024 bit keys is tested. (Version 1.0.3)
4. **Aerys:** Non-blocking HTTP application server. Execution time of Huffman encoding is tested. Huffman coding is the computationally expensive part of the HTTP/2.0 implementation. (Version 0.4.3)

[4] Realized using `php-cgi -T1,N`, which measures `N` executions and discards the first to exclude compilation time.

Application / Library	Speedup	Speedup (inlining)
WordPress	(2.2 ± 0.0) %	(3.5 ± 0.0) %
MediaWiki	(0.8 ± 0.2) %	(1.1 ± 0.2) %
phpseclib RSA enc/dec	(17.9 ± 0.1) %	same
Aerys Huffman enc	(8.0 ± 0.1) %	same

Table 1. Benchmark results for applications and library components.

The WordPress and MediaWiki applications are highly representative of typical PHP web workloads. The phpseclib and Aerys libraries have been chosen as representatives of computationally expensive code as it appears in practical settings. This type of code is uncommon in web-facing code, but appears in back-end processing tasks.

The observed speedup for the different cases is shown in Table 1. Without inlining, web applications see an improvement of 1-2%, while for computationally expensive libraries it is 8-18%. Use of inlining has no effect on the libraries, while WordPress and MediaWiki both experience a slight additional improvement.

The two libraries have been included in the optimization breakdown in Figure 9, where is can be seen that copy propagation, as well as assignment and arithmetic type specialization are responsible for these improvements. Because the execution time difference for WordPress and MediaWiki is small, it is hard to obtain an accurate runtime optimization breakdown for this case. Instead, we may consider static optimization statistics, as shown in Table 2. These statistics refer to all instructions that have been compiled, but not necessarily executed.

From these statistics, it is evident that for both WordPress and MediaWiki the most effective optimization is specialization of assignments and argument sends (specializing ∼7% of all instructions). This is not entirely surprising, as assignments and sends are both a very common instruction type, and their specialization does not require precise type knowledge. DCE, constant propagation and copy propagation only become effective if inlining is en-

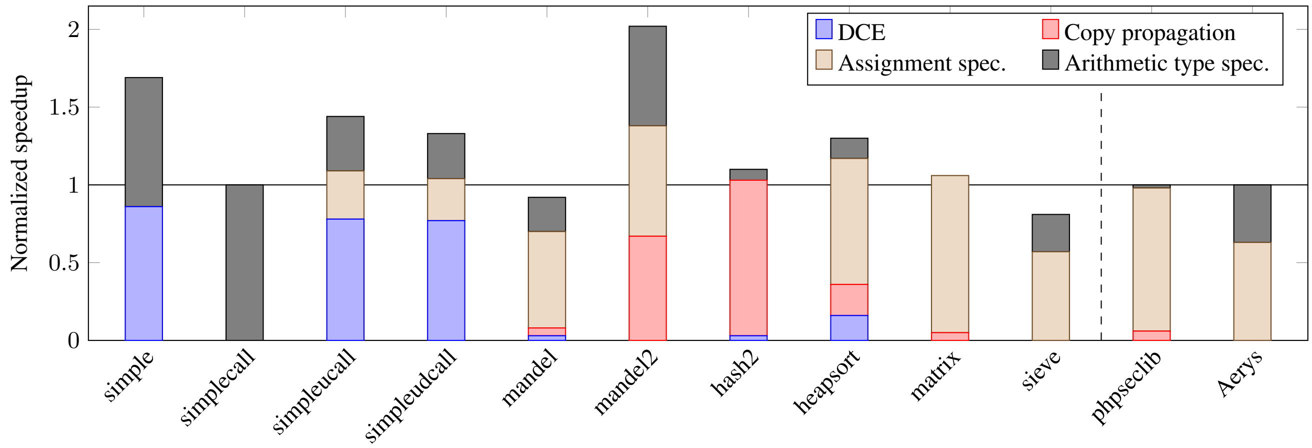

Figure 9. Effect of individual optimizations on micro-benchmarks and libraries. The black line represents the performance improvement if all optimizations are enabled, while the stacked bar charts show the impact of individual optimizations. The individual parts do not necessarily sum to one, if one optimization improves another or if there is overlap between different optimizations. Inlining is enabled in all cases.

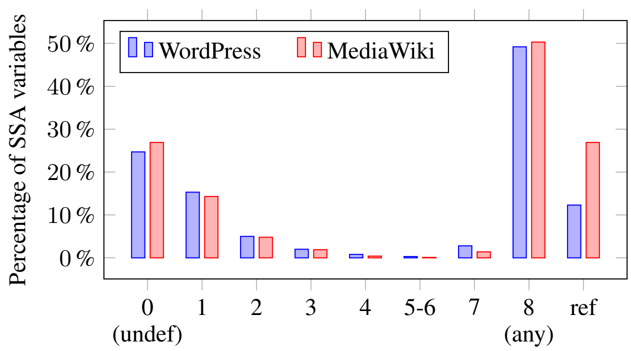

Figure 10. Percentage of SSA variables (CV only), whose inferred type is a union of n types. $n = 0$ implies an always undefined variable, $n = 1$ exact type knowledge and $n = 8$ no type knowledge. Additionally, *ref* is the fraction of variables that may be references.

abled, in which case they account for 9% of eliminated instructions for WordPress, compared to 1% if inlining is disabled.

Table 2 also includes type inference statistics for SSA variables that correspond to compiled variables (non-temporaries). In particular, the percentage of variables for which the inferred type union contains a certain number of types is shown, as well as how many variables are detected as potential references. A graphical version of these results is presented in Figure 10. There are a number of observations we can make:

Firstly, the type distribution is relatively similar for WordPress and MediaWiki. Secondly, approximately 25% of all variables have no type at all. This implies that these variables are *always* undefined. The number of such variables is large, because an implicit always-undefined definition is created in the entry block for each CV variable to ensure strict SSA form. Thirdly, the type distribution is bimodal, with approximately 15% of all variables having exactly one type (i.e., perfect type knowledge), while 50% have 8 types (i.e., no type knowledge).

7. Discussion and Future Work

The results of our experimental evaluation may be briefly summarized as an average speedup of 50% on micro-benchmarks, 13% on computationally intensive library code and 2.3% on typical web applications. Clearly all three categories feature very different performance characteristics. Micro-benchmarks are mostly arithmetic, while applications commonly work on strings and arrays, and have large I/O components. For this reason some optimizations, such as specialization of arithmetic instructions, have a major impact on micro-benchmark performance, but play only a minor role in the optimization of web applications. In the following, we will discuss some of the limiting factors of our approach and how they might be overcome. For this we focus on the web application case, as it is the practically most important one.

First of all, type inference clearly plays a very important role in the static optimization of dynamic languages, both because type specialization is an important class of optimizations, but also because nearly all other code transformations require some degree of type information for correctness. For micro-benchmarks we are commonly able to precisely determine the type of inner-loop variables and fully exploit this type information through specialization. For large web applications this is not the case. As Figure 10 illustrates, we do not have any type knowledge for approximately half of all SSA variables.

There are a number of reasons for this. One issue is the single-file compilation view that is enforced by opcache, as discussed in section 3.8. Due to this limitation, we do not know any signatures for functions defined outside the current file and have to pessimistically assume by-reference argument passing. Any reference variable must be assumed to have any type, and variables derived from references are likely to inherit this property.

We believe that removing this single-file limitation is important to advance optimization in PHP, both for inference and other purposes. A more aggressive approach is the introduction of something akin to HHVM's RepoAuthoritative mode [45], which requires all code to be known in advance and forbids certain runtime operations. This has the advantage that *all* symbols are known during compilation. For example, such a mode can guarantee that a certain virtual method will never be overridden, so the exact callee is known. On the other hand, RepoAuthoritative mode is known to significantly complicate the deployment process.

Another limiting factor is that our type inference is function-local: no inter-procedural inference is performed. It would be possible to use the Cartesian product algorithm [21] (or a faster variant thereof) to extend inference across procedure boundaries. However, applicability would be limited due to both the single-file limitation and the uncertainty about the callee of virtual method calls.

Metrics	WordPress		WordPress (inlining)		MediaWiki		MediaWiki (inlining)	
Instrs	103770		199149		124591		167137	
Eliminated instrs (const prop.)	406	(0.4%)	11013	(5.5%)	151	(0.1%)	1109	(0.7%)
Eliminated instrs (DCE)	155	(0.1%)	4756	(2.4%)	147	(0.1%)	3173	(1.9%)
Eliminated instrs (copy prop.)	487	(0.5%)	2198	(1.1%)	421	(0.3%)	1888	(1.1%)
Specialized instrs (arithmetic)	69	(0.1%)	164	(0.1%)	99	(0.1%)	184	(0.1%)
Specialized instrs (assignment)	3873	(3.7%)	9664	(4.9%)	5404	(4.3%)	8222	(4.9%)
Specialized instrs (arg. send)	3712	(3.6%)	5686	(2.9%)	3251	(2.6%)	3983	(2.4%)
SSA variables (CV only), of which	34393		88685		47274		73866	
... may be reference	8856	(12.3%)	18513	(20.9%)	12723	(26.9%)	16905	(22.9%)
... may be refcounted	24337	(70.8%)	50549	(57.0%)	31673	(67.0%)	44149	(59.8%)
... have 0 types (undef)	8506	(24.7%)	31070	(35.0%)	12722	(26.9%)	24861	(33.7%)
... have 1 type	5405	(15.7%)	12584	(14.2%)	7140	(15.1%)	11593	(15.7%)
... have 2 types	1562	(4.5%)	3495	(3.9%)	1901	(4.0%)	3017	(4.1%)
... have 3 types	681	(2.0%)	1267	(1.4%)	893	(1.9%)	1464	(2.0%)
... have 4 types	265	(0.8%)	768	(0.9%)	170	(0.4%)	325	(0.4%)
... have 5-6 types	88	(0.3%)	396	(0.5%)	24	(0.1%)	56	(0.1%)
... have 7 types	952	(2.8%)	1827	(2.1%)	663	(1.4%)	1061	(1.4%)
... have 8 types (any)	16931	(49.2%)	34381	(38.8%)	23758	(50.3%)	31306	(42.4%)
Compile time increase	59 ms	(50%)	140 ms	(120%)	75 ms	(37%)	116 ms	(58%)

Table 2. Static optimization and type inference statistics for WordPress and MediaWiki, with and without inlining. The SSA variable type statistics include only compiled variables (CVs), temporary variables are excluded.

We expect that the use of type annotations (section 3.7) will increase as code-bases move to support PHP 7 only, and that type declaration support will be expanded to more language items, including object properties. This will provide more type roots for inference, and in particular alleviate our inability to infer property types (section 3.5).

However, as PHP is a dynamic language, there will always remain value sources for which no type is known. One way to approach this problem is the use of speculation: given a *likely* (but not guaranteed) type for a value source, we can generate two code-paths, one specializing on the chosen type, the other acting as a fallback. The likely type may be determined either statically, based on the usage of the value, or dynamically, using runtime type feedback. An obstacle to a purely static approach is that, without information about hot functions, such speculation might lead to a large code size increase.

Next to the difficult problem of inferring accurate types in a dynamic language, another significant constraint on static bytecode optimization is imposed by the need to operate within the limitations of the used virtual machine. It is not possible to perform overly fine-grained specialization, as this leads to excessive VM code size growth. Instead, we have to focus on specific instructions and type combinations that promise to be have the largest effect. Similarly, splitting instructions in order to allow separate optimization of the individual parts, is only possible to a limited degree, because the additional instruction dispatch overhead quickly overshadows any benefit. Indeed, the reverse process of using superinstructions (which combine multiple instructions into a single one), is a well-known interpreter optimization [26].

As these problems relate to VM overhead, they can ultimately only be overcome with a JIT compiler. Work on a new JIT compiler for PHP [46], which is based on the SSA-based optimization framework introduced in this work, is already underway.

8. Conclusion

In this work we have investigated the applicability of purely static, transparent, bytecode-level optimizations to the dynamic programming language PHP. To this purpose an SSA-based optimization infrastructure was used, in combination with a type inference algorithm based on SCCP. Implemented optimizations include type specialization, constant and copy propagation, dead code elimination and inlining. Our experimental evaluation has shown an average speedup of 50% on micro-benchmarks, 13% on computationally intensive libraries, as well as 1.1% (MediaWiki) and 3.5% (WordPress) on web applications.

As such, we have demonstrated that static optimization techniques can yield significant improvements even when applied to a dynamic language. However, these improvements heavily depend on the characteristics of the application, with computationally intensive code optimizing much better than typical web applications.

The described optimization framework and the specialization-based optimizations will be part of PHP 7.1. The remaining optimizations depend on inlining to become effective, which requires further work prior to wide usage (e.g., backtrace preservation). For this reason, these optimizations target a later version of PHP and are currently maintained in a fork [47].

Acknowledgments

This work is based on several components initially implemented as part of an experimental JIT compilation project developed by Zend Technologies.

References

[1] W3 Web Technology Surveys. Usage of server-side programming languages for websites, 2016. URL `https://w3techs.com/technologies/overview/programming_language/all`. Retrieved: Nov. 1, 2016.

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

[3] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop virtual machine. In *OOPSLA'14*, pages 777–790. ACM, 2014.

[4] Paul Bissonette. Lockdown results and HHVM performance. HHVM blog, June 2015. URL `http://hhvm.com/blog/9293/`

lockdown-results-and-hhvm-performance. Retrieved: Dec. 28, 2016.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[6] Free Software Foundation. The GCC internals documentation. 12.3 static single assignment, 2016. URL https://gcc.gnu.org/onlinedocs/gccint/SSA.html. Retrieved: Nov. 1, 2016.

[7] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

[8] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, pages 75–86. IEEE Computer Society, 2004.

[9] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.

[10] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Softw. Pract. Exper.*, 27(6):701–724, June 1997.

[11] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC'06*, pages 247–262. Springer Berlin Heidelberg, 2006.

[12] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *POPL'98*, pages 107–120. ACM, 1998.

[13] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC'96*, pages 253–267. Springer-Verlag, 1996.

[14] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI'00*, pages 321–333. ACM, 2000.

[15] Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. A verifiable SSA program representation for aggressive compiler optimization. In *POPL'06*, pages 397–408. ACM, 2006.

[16] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI'13*, pages 175–186, 2013.

[17] Delphine Demange, David Pichardie, and Léo Stefanesco. Verifying fast and sparse SSA-based optimizations in coq. In *CC'15*, pages 233–252, 2015.

[18] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. Verified construction of static single assignment form. In *CC'16*, pages 67–76, 2016.

[19] Delphine Demange and Yon Fernandez de Retana. Mechanizing conventional SSA for a verified destruction with coalescing. In *CC'16*, pages 77–87, 2016.

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

[21] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP'95*, pages 2–26. Springer-Verlag, 1995.

[22] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *CGO'10*, pages 278–287. ACM, 2010.

[23] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *DLS'12*, pages 73–82. ACM, 2012.

[24] Stefan Brunthaler. Inline caching meets quickening. In *ECOOP'10*, pages 429–451. Springer-Verlag, 2010.

[25] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, June 1973.

[26] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6), October 2007.

[27] Stefan Brunthaler. Interpreter instruction scheduling. In *CC'11*, pages 164–178. Springer Berlin Heidelberg, 2011.

[28] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The HipHop compiler for PHP. In *OOPSLA'12*, pages 575–586. ACM, 2012.

[29] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.

[30] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *SAC'09*, pages 1916–1923. ACM, 2009.

[31] Jan Benda, Tomas Matousek, and Ladislav Prosek. Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. In *Proc. on the 4th International Conference on .NET Technologies*, pages 11–20, 2006.

[32] Peachpie PHP compiler to .NET, 2016. URL https://github.com/iolevel/peachpie. Retrieved: Nov. 1, 2016.

[33] Quercus: PHP in Java. URL http://quercus.caucho.com/quercus-3.1/doc/quercus.xtp. Retrieved: Nov. 1, 2016.

[34] JPHP - an alternative to PHP. URL http://j-php.net/. Retrieved: Nov. 1, 2016.

[35] HippyVM - an implementation of the PHP language in RPython. URL https://github.com/hippyvm/hippyvm. Retrieved: Dec. 28, 2016.

[36] Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. In *POPL'09*, pages 200–212. ACM, 2009.

[37] Nikita Popov. Forbid dynamic calls to scope introspection functions. PHP RFC proposal, May 2016. URL https://wiki.php.net/rfc/forbid_dynamic_scope_introspection. Retrieved: Nov. 1, 2016.

[38] Joe Watkins and Phil Sturgeon. Typed properties. PHP RFC proposal, March 2016. URL https://wiki.php.net/rfc/typed-properties. Retrieved: Nov. 1, 2016.

[39] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm, 2001. URL https://www.cs.rice.edu/~keith/EMBED/dom.pdf.

[40] Victor H. S. Campos, Raphael E. Rodrigues, Igor R. de Assis Costa, and Fernando M. Q. Pereira. Speed and precision in range analysis. In *SBLP'12*, pages 42–56. Springer-Verlag, 2012.

[41] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

[42] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-SSA translation for correctness, code quality and efficiency. In *CGO'09*, pages 114–125. IEEE Computer Society, 2009.

[43] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO'08*, pages 35–44. ACM, 2008.

[44] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *POPL'94*, pages 397–408. ACM, 1994.

[45] HHVM documentation. Advanced usage: Repo authoritative. URL https://docs.hhvm.com/hhvm/advanced-usage/repo-authoritative. Retrieved: Nov. 1, 2016.

[46] Dmitry Stogov. JIT for PHP project. PHP internals list. URL http://news.php.net/php.internals/95531. Retrieved: Nov. 1, 2016.

[47] PHP static optimization fork. URL https://github.com/nikic/php-src/tree/opt.