Bachelor Thesis

Optimizing PHP Bytecode using Type-Inferred SSA Form

Nikita Popov

Matriculation Number: 347863

Technische Universität Berlin
Faculty IV · Electrical Engineering and Computer Science
Department of Computer Engineering and Microelectronics
Embedded Systems Architectures (AES)
Einsteinufer 17 · D-10587 Berlin

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science

according to the examination regulations at the Technische Universität Berlin for the Bachelor in Computer Science of May 28, 2014.

Department of Computer Engineering and Microelectronics
Embedded Systems Architectures (AES)
Technische Universität Berlin
Berlin

Author Nikita Popov

Thesis period February 16, 2016 to July 5, 2016

Referees Prof. Dr. B. Juurlink, Embedded Systems Architectures (AES)
Prof. Dr. S. Glesner, Software Engineering for Embedded Systems

Supervisor Dr. B. Cosenza, Embedded Systems Architectures (AES)

Declaration

According to §46(8) of the Examination Regulations

I hereby confirm to have written the following thesis on my own, not having used any other sources or resources than those listed.

Berlin, July 4, 2016

Nikita Popov

Abstract

PHP is a dynamically typed programming language, which is commonly used for the server-side implementation of web applications. As such its performance is often critical to the response time, throughput and resource utilization of such applications.

This thesis aims to reduce runtime overhead by applying classical data-flow optimizations to the PHP bytecode in static single assignment form. Type inference is used both to enable the use of type-specialized instructions and to ensure the correctness of other optimizations, which are commonly only applicable to certain types. Next to type-specialization, we also implement flow-sensitive constant propagation, dead code elimination and copy propagation. Additionally, inlining is used to increase the applicability of other optimizations.

The main challenge is to reconcile classical compiler optimizations, that have been developed in the context of statically typed and compiled languages, with a programming language that is not only dynamically and weakly typed, but also supports a plethora of other dynamic language features. This requires a careful analysis of language semantics and modification of standard algorithms to support them.

Our approach results in significant performance gains for numerically intensive and tightly looped code, as is typically found in benchmark scripts. We achieve a mean speedup of 1.42× on PHP's own benchmark suit. However, when considering real applications we have found the speedup to be limited to 1-2%.

Zusammenfassung

PHP ist eine dynamisch typisierte Programmiersprache, welche häufig für die Serverseitige Implementierung von Web-Applikationen genutzt wird. Aus diesem Grund ist die Effizienz der PHP-Implementierung kritisch für die Ausführungszeit, den Durchsatz und den Ressourcen-Verbrauch derartiger Applikationen.

Ziel dieser Bachelorarbeit ist es die Performanz der PHP-Implementierung zu verbessern, indem klassische Datenfluss-Optimierungsmethoden auf den PHP Bytecode in Single-Static-Assignment Form angewendet werden. Typ-Inferenz wird genutzt, sowohl um die Nutzung von Typ-spezialisierten Instruktionen zu ermöglichen, als auch um die Korrektheit anderer Optimierungen sicherzustellen, welche oftmals nicht auf alle Typen anwendbar sind. Neben Typ-Spezialisierung wurde auch Kontrollfluss-sensitive Propagation von Konstanten, Elimination von totem Code, sowie Propagation von Kopien umgesetzt. Zusätzlich wird durch Inline-Ersetzung von Funktionen die Anwendbarkeit anderer Optimierungen erhöht.

Hierbei ist die hauptsächliche Herausforderung, dass klassische Compiler-Optimierungsmethoden im Kontext von statisch typisierten und kompilierten Sprachen entwickeln wurden, während PHP nicht nur dynamisch und schwach typisiert ist, sondern auch eine Vielzahl anderer dynamischer Sprachelemente unterstützt. Dies erfordert eine sorgfältige Analyse der Sprachsemantik und Anpassung von Standard-Algorithmen, um diese zu unterstützen.

Unsere Herangehensweise führt zu signifikanten Verbesserungen in der Ausführungszeit von numerisch intensivem Code, wie er typischerweise in Benchmarks gefunden wird. In PHP's eigener Benchmark-Suite verringert sich die Ausführungszeit im Durchschnitt um einen Faktor von 1,42. Für realistische Applikationen begrenzt sich die Verbesserung jedoch auf 1-2%.

Contents

List of Figures

List of Listings

1. Introduction

Dynamic scripting languages are commonly chosen over classic statically typed languages, because their use of dynamic typing and lack of an explicit compilation step enables a higher degree of productivity. Unfortunately, the same features that make these languages productive, also make them hard to implement efficiently: In order to support their dynamic features, scripting languages are traditionally implemented using interpreters.

An increasingly common avenue used to improve the performance of such languages, is the employment of a just-in-time (JIT) compiler, which generates native machine code at runtime. However, the implementation of JIT compilers is not only a major feat of engineering, but also carries a significant increase in implementation complexity. This thesis pursues a different approach, namely the use of classical data-flow optimization techniques to improve the quality of the interpreted bytecode.

We implement our performance optimizations for the PHP programming language, which is commonly used for the server-side implementation of web applications. PHP powers both some of the largest websites such as Facebook[1], Wikipedia and Yahoo, but also countless small websites, like personal blogs, discussion forums, etc.

End-to-end web application performance depends on more factors than only the performance of the server-side programming language, in particular it also includes network transmissions, processing of database queries and client-side rendering. Nonetheless PHP's performance plays an important role in determining the response time, throughput and resource utilization of web applications.

Our approach to optimization is to reduce runtime overhead by applying classical data-flow optimizations, such as constant propagation, dead code elimination and copy propagation, to the PHP bytecode in static single assignment (SSA) form. Type inference is used both to enable the use of type-specialized instructions and to ensure the correctness of other optimizations, which are often only applicable to certain types. We also experiment with the impact of inlining on other optimizations.

The primary challenge we face, is that these classical optimization techniques have been developed in the context of statically typed and compiled languages, as such their application to a programming language that is not only dynamically and weakly typed, but also supports a plethora of other dynamic language features, requires special consideration.

Unlike many alternative scripting language implementations, which chose to omit certain little used and particularly hard to optimize language features, our optimizations are implemented as an extension of the reference implementation, and as such need to support the full scope of the language. However, we did submit a number of language change proposals for the removal of some particularly problematic edge-cases as part of

[1] Facebook uses the Hack programming language, which is derived from PHP.

this work.

To summarize our results, we have found that significant performance gains are possible for numerically intensive and tightly looped code, as is typically found in benchmark scripts. We achieve a mean speedup of 1.42× on PHP's own benchmark suit. However, when considering real applications we have found the speedup to be limited to 1-2%. The reason for this is in part that we cannot derive sufficient static information for realistic applications, however a limitation of the current compiler, which requires all files to be compiled independently, also plays a significant role.

1.1. Historical context

This thesis is part of a larger effort to improve the performance of the reference PHP implementation. For many years performance of the PHP implementation has stagnated, with optimization being limited to small incremental improvements.

To reduce the resource utilization of their own PHP deployment, Facebook implemented first the HipHop compiler [37], which compiles PHP code to C++, and later the HipHop Virtual Machine (HHVM) [1], which makes use of a just-in-time compiler. Both of these alternative PHP implementations have shown that it is possible to significantly improve the performance of real-world PHP applications, while maintaining a high degree of compatibility with the reference implementation.

In part in response to these developments, Zend Technologies also started an attempt at implementing a JIT compiler [30] for PHP, using LLVM for code generation. However, this compiler was both extremely slow and did not show any performance advantage on real applications. It was concluded that other parts of the PHP implementation needed to be improved before a JIT could be worthwhile.

As such, an effort was started to redesign all core data structures to reduce memory usage, indirection and allocation, and make them generally more CPU cache efficient. This effort, together with many other optimizations, eventually resulted in a new major version, PHP 7. For many applications PHP 7 improved throughput by a factor of two, while also significantly reducing memory utilization. This made performance of PHP 7 competitive with HHVM, even though HHVM still holds an advantage if a precompiled code repository is used.

Because PHP 7 already implemented most of the low-hanging optimization fruit, further performance improvements require the pursuit of different avenues. One possibility it to revisit the use of a JIT compiler. Another is to improve the implementation of the virtual machine and the used instruction format. Another, which is pursued in this thesis, is the use of compiler optimizations to improve bytecode quality.

1.2. Prior work

In the following we will provide a short overview of related prior work, focusing mostly on the use of static analysis techniques, and in particular of type inference, to improve

the performance of dynamic languages. As such, we will usually not cover just-in-time compilers, but include some static analysis work not related to optimization. We will first cover prior work in other dynamic languages and then, in more detail, prior work applying directly to PHP.

In other dynamic languages The Starkiller project [29] compiles Python code into C++. It uses an augmented Cartesian Product Algorithm (CPA) [2] for type inference. The basic idea of the CPA algorithm is that for a given function call, the cartesian product of the possible types of the arguments is computed, resulting in a set of monomorphic (single-type) argument type lists. The result type of the call is then the union of the result types for the individual monomorphic argument type combinations.

Cannon [13] extended the Python interpreter with local type inference based on iterative type analysis [14], in order to allow the use of type-specialized instructions. Unlike the Starkiller project, no language restrictions were imposed in this case. However, Cannon concluded that the observed speedup (1% even for microbenchmarks) is not worthwhile.

The Diamondback Ruby (DRuby) project [20] extends the Ruby programming language with support for type annotations, which in conjunction with type inference, can be used to discover type errors. The PRuby project [19] extends DRuby by the ability to gather runtime information about use of dynamic language features through instrumentation, which is then used to replace dynamic features with statically analyzable alternatives.

Jensen et al. [23] present a static analysis infrastructure for JavaScript, which performs type inference and points-to analysis based on abstract interpretation and the monotone framework. This work is not targeted at optimization, but rather wishes to provide type information for program comprehension and tooling. Hacket and Guo [22] describe how unsound type inference can be combined with dynamic type updates that are performed if unlikely situations are hit during evaluation of just-in-time compiled code.

An interesting approach to incorporating type information during execution is dynamic interpretation [34], which interprets programs based on a flow graph, which directs program flow not only based on control flow, but also on type changes. The flow is split at all points where type uncertainly exists, thus allowing the use of fully type-specialized instructions on the individual branches.

Würthinger et al. [36] investigate the use of abstract syntax tree (AST) based interpreters that incorporate type feedback at run-time. The rationale for using an AST-based interpreter over a more commonly used bytecode interpreter, is that ASTs are more malleable: It is easy to modify an AST at run-time to incorporate type information, while bytecode modifications may require more involved updates of jumps offsets. Brunthaler [12] approaches the problem of dynamic bytecode updates by adding an additional inline cache pointer to each instruction. This cache pointer can then updated at run-time to reference a specialized implementation of the operation.

In PHP The undoubtedly most practically significant alternative PHP runtime is the HipHop Virtual Machine (HHVM) [1], which not only managed to improve performance significantly relative to PHP 5, but also maintains language parity to a high degree.

HHVM uses a just-in-time compiler, which operates on tracelets, which are regions of code with a single entry but potentially multiple exits. The tracelet is symbolically executed in a single-pass, forward data-flow analysis, which annotates instructions with input and output types where possible. Input types that could not be statically determined are instead observed at runtime. For a set of observed types, type guards are inserted at the start of the tracelet, allowing the application of classic compiler optimizations based on mostly complete type information. If a type guard fails, the tracelet is compiled with another set of input types and the guards are rewired to jump to the new tracelet. This can be repeated to a certain limit, allowing the handling of mildly polymorphic code (however, the vast majority of code has been observed to be monomorphic).

The precursor of HHVM is the HipHop compiler (HPHPc) [37], which compiles PHP code to C++. The compiler infers types based on an adaptation of the Damas-Milner constraint-based algorithm [18] and specializes the generated code based on the results. All operations are performed directly on the abstract syntax tree (with control flow graph), no bytecode representation is used. HPHPc imposes a number of restrictions on the language, and exploits them for optimization, most notable of which are the requirement that all code must be known in advance and that integer arithmetic does not overflow to double.

The work that is likely most closely related to what is attempted in this thesis, is Paul Biggar's dissertation on the implementation of the PHP to C compiler phc [5]. As phc was implemented prior to PHP 5.4, where support for call-time pass by reference was removed, a large focus of this work is on modeling the aliasing behavior of references. Hashed SSA form [15] is used, which extends SSA form by additional μ (may-use) and χ (may-define) nodes. Alias analysis, type inference and constant propagation are performed simultaneously and *before* construction of SSA form.

We do not follow this methodology, as we no longer need to consider call-time pass by reference, and we further believe that the alias analysis results are only correct if PHP's error handling mechanism is ignored. While phc intentionally did not model error handling, this option is not available to us. As such, there is an overlap with the general analysis passes we implemented in this thesis (such as type inference and constant propagation), however both the methods that are used and the general architecture are different.

The dynamic nature of PHP makes it hard for an ahead-of-time compiler to generate efficient code, while at the same time implementing a just-in-time compiler from scratch requires a major feat of engineering. For this reason a number of alternative PHP implementations attempt to leverage existing JIT compilers instead: Phalanger [4] and its successor Peachpie [38] target the Common Language Runtime (CLR, the .NET runtime), while Quercus [39] and P9 [31] target the Java Virtual Machine (JVM). As both the CLR and JVM are statically typed virtual machines, compilation for them has many of the same challenges as compilation to C or C++. The hope is that their JIT compilers are in a better position to optimize the generated bytecode.

Finally, there are a number of instances where SSA form has been applied to PHP code for the purpose of static analysis rather than optimization. One example is Minamide's work on the "static approximation of dynamically generated web pages" [25], which uses SSA form to construct an approximate context free grammar for the output of a program,

for use in security analysis and verification. Another is Rimsa et al. [28], who perform efficient taint analysis using a sparse SSA-based algorithm. In particular, they use e-SSA form [6] to improve control flow sensitivity, which is a modification of SSA form we also employ in a different capacity.

1.3. Attribution

This work is based on several components implemented as part of Zend's experimental JIT project [30]. In particular the implementation of static single assignment (SSA) form, as well as value range and type inference are reused. The main author of these components is Dmitry Stogov, the lead engineer of the PHP project.

These components will be described in varying detail, depending on how important they are for further considerations and how much we had to work with and on them. Type inference is described in detail, both because it plays an important role for other analysis passes and because we have significantly departed from the original implementation. Apart from countless correctness fixes, we switched to a different, control flow sensitive algorithm, extended inference to use π-nodes and implemented a new type narrowing algorithm, as the previous one had correctness issues.

The SSA implementation is also discussed extensively, because it is the basis of all further work. However, in this case we will describe the construction algorithms only in broad strokes, as we did not modify them apart from various correctness and performance fixes. Only the placement of π-nodes is discussed in more detail, as more significant changes were necessary to ensure correctness in this case.

We do not discuss value range inference, as we had little contact with it and, while it does contribute to the quality of the type inference results, an understanding of value range inference is not of particular importance to the remainder of the work.

We mention this upfront to avoid creating the impression that *everything* discussed in this thesis has been implemented by us.

1.4. Outline

The remainder of this thesis is split into five chapters, for which a brief overview will be provided here:

Chapter 2 discusses semantics of the PHP programming language insofar as they are relevant to optimization. This chapter illustrates many of the issues that make optimization of PHP challenging and also motivates some of the design choices we have made for our optimization passes.

Chapter 3 introduces a number of prerequisites for the primary optimization work. An overview of the PHP execution model and virtual machine are provided. More importantly, static single assignment form and PHP specific modifications to it are introduced in this chapter.

Chapter 4 subsequently presents the analysis and optimization passes that have been

implemented. This includes descriptions of a general data-flow propagation framework, as well as type inference and constant propagation based on it. Other optimizations that are described include dead code elimination, copy propagation and type specialization.

Chapter 5 then presents the results of this work, by considering the performance impact on both microbenchmarks and real applications. The impact of individual optimizations is analyzed, and the reasons why certain optimizations do or do not apply is discussed.

Chapter 6 provides a conclusion and outlook. Finally, Appendix A contains instructions for obtaining the source code developed for this thesis.

2. PHP Language Semantics

This chapter aims at providing an overview of PHP language semantics, insofar as they are relevant for optimization. We will dispense with a description of general language syntax, as it is sufficiently similar to other C-like languages and not particularly relevant for our purposes.

Instead we will focus on covering various edge-case semantics that may be problematic for performing optimizations. Based on these, we will also motivate some of the choices we have made in our approach to optimization and explain why certain avenues were not pursued.

2.1. Dynamic and weak typing

PHP is a dynamically typed language, which means that types of variables are generally only determined at runtime and even then 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, the operation `"2" * "5"` evaluates to integer 10, with the two strings being implicitly cast to integers before the multiplication is performed. Following the same logic, the operation `"foo" * "bar"` evaluates to integer zero, because the cast of non-numeric strings to integer is defined as zero (since PHP 7.1 a run-time warning is emitted for this case).

Additionally it is common for basic operations to return different result types depending not only on the types but also specific values of the input operands:

```
var_dump(1 + 1);                    // int(2)
var_dump(1.0 + 1.0);                // float(2.0)
var_dump(PHP_INT_MAX + 1);          // float(9.2233720368548E+18)
var_dump(['a' => 1] + ['b' => 2]); // ['a' => 1, 'b' => 2]
```

This example illustrates that an addition operation may evaluate to either an integer, a floating-point number or an array[1]. A float result is possible not only if one of the input operands is a float, but also if the result of an integer addition overflows. This kind of result type overloading makes it hard to statically infer types.

[1]The "array" type in PHP is an ordered dictionary implemented using a hashtable. The language makes no strong distinction between vectors and dictionaries.

2.2. References

PHP supports a concept of references, which are in many ways similar to pointers in C or references in C++. Consider this example comparing use of references in PHP with roughly equivalent C++ code:

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

```cpp
int var1 = 42;
int& var2 = var1;
var2 = 24;
printf("%d\n", var1); // 24
```

However, there are some differences to note: Firstly, references in PHP are fully symmetric, there is no concept of a directional "reference to" between two variables. The assignment `$var2 =& $var1` will not only make `$var2` a reference to `$var1`, but also `$var1` a reference to `$var2`, or more accurately, both variables will be references to a hidden shared value and can be used completely interchangeably henceforth.

Secondly, C++ requires reference variables to be explicitly declared, while in PHP a non-reference variable can be converted into a reference variable through some form of reference assignment. In particular this can also be control flow dependent. In some cases, most notably the by-reference argument passing discussed below, it may even be impossible to statically determine whether a variable holds a reference.

One particularly problematic aspect of references is their behavior when used as array elements, in which case the reference will be preserved even if the array undergoes a by-value copy or argument pass. For instance:

```php
$value = 42;
$array = [&$value]; // create array containing reference
$copy = $array;     // by-value copy
$copy[0] = 24;
var_dump($value);   // int(24)
```

In this example `$copy = $array` performs a by-value copy of the array, but nonetheless the `$copy[0]` element remains a reference. For example this implies that a write into an array that is subsequently not used cannot be eliminated as dead code, unless it is known that the written element cannot be a reference.

Apart from reference assignments using `=&`, there are a number of other ways in which references can be created, the most important one being by-reference argument passing:

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

Here the function call will create a reference between the `$n` function argument and the `$i` local variable. Notably, there is no indication that by-reference argument passing is being used on the side of the caller.

Because the PHP compiler is, at least currently, limited to compiling each file separately, it is very common that the signature of the called function (defined in a different file) is not known. In this case we have to pessimistically assume that by-reference argument passing is being used, even though this is most likely not the case. Due to reasons outlined in section 2.5, it is not feasible to perform optimizations on variables that are potential references.

The combination of these two factors implies that any variable that is passed to an unknown function is effectively excluded from optimization from that point forth. This is one of the most significant issues when optimizing PHP code within the existing compiler framework.

2.3. 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. In compiler terminology, a variable assignment constitutes a definition of the variable, but not a use. In the general case, this is not true in PHP for three reasons, which will be illustrated in the following.

The first and practically most important one is that the assigned-to variable may hold a reference, in which case the modification happens not to the variable itself, but rather to the content of the reference. Consider the example from the previous section, this time using a C pointer analogy:

```
$var1 = 42;
$var2 =& $var1;
$var2 = 24;
```

```
int var1 = 42;
int *var2 = &var1;
*var2 = 24;
```

In this case the last assignment `$var2 = 24` must know the value of `$var2` in order to change the value of the reference. The nature of this assignment is particularly clear in the analogous C code, because it contains an explicit dereferencing operation: `*var2 = 24` only reads `var2`, but does not write to it. The actual write is performaned on the dereferenced value only.

The second problematic case occurs when the variable is the last user of an object that has a destructor with observable side-effects:

```
class Dtor { function __destruct() { echo "Dtor\n"; } }
$var = new Dtor;
$var = 42;
```

In this instance the last assignment not only modifies the value of `$var`, but also destroys its old value, thus triggering the object destructor. As PHP uses reference-counting, the destruction order is deterministic[2] as long as no circular references are involved, and as such precise destruction semantics should be maintained during program transformations.

[2]Unlike C++, PHP does not have a strictly specified destruction order. However it is generally understood that (non-circular) destruction should occur as soon as possible.

Lastly, objects implemented by PHP extensions may provide a `set` handler, which can be used to overload the assignment operator. None of the core extensions use this feature, but it needs to be taken into account in order to support third-party extensions.

Due to these three cases we have to assume, at least in absence of additional type information, that any assignment acts both as a use and definition of the variable. This has significant implications on the structure of the SSA graph and transformations acting on it.

2.4. Dynamic scope introspection and modification

A somewhat peculiar feature supported by PHP is syntactic support for performing dynamic scope introspection through *variable variables*:

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

In this example `$varName` stores the name of a different local variable `$var` and then indirectly references it using the `${$varName}` syntax. If `$varName` were a dynamic value rather than a static string, the assignment through `${$varName}` could modify any variable in the current scope, making this language feature inherently hard to predict.

There exist a number of other ways in which the local scope may be dynamically read or modified: For example, PHP allows executing arbitrary code using the `include` and `eval` directives, the former acting on files, the latter on strings. While this is not their primary purpose, these constructs inherit the surrounding scope and may perform arbitrary modifications on it.

Additionally there are a number of functions which can also perform dynamic scope introspection. As an example, the `extract()` function can be used to extract an associative array into local variables. These functions are particularly problematic, because prior to PHP 7.1 there was no reliable way of detecting calls to them, as invocations could also occur dynamically:

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

In this example it would still be possible to detect the dynamic call `$fn()` and assume that it may modify variables. However, functions may also be invoked through various implicit callbacks such as autoloading handlers, which are automatically called in many situations. As the inability to detect such calls without falling back to very pessimistic assumptions would have severely limited the scope of possible optimizations, we have

submitted a proposal to forbid dynamic calls to scope introspection functions [27], which has been accepted for PHP 7.1.

With this change in place, we are able to detect all cases where variables may be modified dynamically. As all the constructs described in this section are only rarely used in practice, we limit ourselves to detecting their uses, but do not try to optimize functions utilizing them. A more fine-grained approach would be to treat such dynamic modifications as potential definition points for all variables in the current scope, however we do not consider the added complexity to be worthwhile.

2.5. Error handling

PHP has three broad categories of errors: Compile-time errors, exceptions and run-time warnings. Compile-time errors cannot be caught and abort execution of the program. Relatively few conditions are enforced at compile-time, as most checks are delayed to run-time instead. Exceptions work the same way as in Java, i.e. they will abort execution of the current control path and unwind the call stack until they are caught.

Run-time warnings[3] will not abort execution of the current control path. The warning will typically be either displayed or logged and some kind of error-indicating value will be returned from the operation. Run-time warnings are problematic for optimization for multiple reasons:

First, the possibility that an error-indicating value may be returned reduces the quality of type inference results. For example, if we have inferred that a certain array only contains integers, we cannot conclude that a read from this array will return an integer: In case the accessed key does not exist, a run-time warning is issued and the return value is null.

Second, it is possible to register an error handling function, which is automatically invoked whenever a run-time warning is generated. Because nearly all operations in PHP can generate run-time warnings in some situation, this means that arbitrary and unknown PHP code can run at nearly any point during the execution of a function. This code could potentially modify any global variable, or more generally any value that has aliases outside the function.

However, the situation is even worse than this, because the error handler is also passed the variable scope in which the error occurred. This allows the error handler to modify references and objects, even if they are local to the function:

```
function test() {
    $obj = new stdClass;
    $obj->prop = 42;
    echo $undef; // trigger run-time warning
    var_dump($obj->prop); // int(24)
}
```

[3] Warnings are only one out of many types of non-fatal run-time errors. For simplicity we refer to all errors of this kind as “warnings”.

```
set_error_handler(function($_1, $_2, $_3, $_4, $scope) {
    $scope['obj']->prop = 24;
});
test();
```

In this example the `$obj` object would normally be considered local to the function, a fact that could be established through escape analysis. However, as the error handler is passed the variable scope, it can still perform arbitrary modifications to object properties and references.

For this reason we do not attempt to show that the value or type of a reference or object property is stable through use of escape and alias analysis: There will always exist the possibility of a modification through the error handler, and for realistic code we generally cannot prove freedom of run-time warnings. This would only become worthwhile if the scope parameter of the error handler could be removed in the future.

2.6. Global variables and the pseudo-main scope

Unlike many other languages, PHP does not implicitly import variables from the global scope into functions. Instead, use of global variables needs to be made explicit by using the `global` keyword or the special `$GLOBALS` dictionary:

```
function incGlobal() {
    global $var; /* or */ $var =& $GLOBALS['var'];
    $var++;
}
$var = 1;
incGlobal();
var_dump($var); // int(2)
```

This strict separation comes to our advantage, as we can easily detect variables that refer to globals and not perform optimizations on them (which are not possible due to the issues described in the previous section).

A PHP file can, next to declarations for functions, classes and so on, also contain free-standing code, referred to as *pseudo-main* code. It should be noted that the pseudo-main scope of a file does *not* necessarily coincide with the global scope. Rather, the pseudo-main code will adopt the scope of the location it is included in, which may be the global scope, but may also be a local function scope, if the file is included inside a function body.

This scope adoption makes it essentially impossible to perform optimizations on pseudo-main code. The following example illustrates how the result of even very simple code can be drastically altered if the scope is initialized appropriately before the file inclusion:

```
// file1.php
$a = 1;
$b = 1;
```

```php
var_dump($a + $b); // int(11) if run through file2.php

// file2.php
class Dtor { function __destruct() { $GLOBALS['b'] = 10; } }
$b = new Dtor;
include 'file1.php';
```

On the assignment `$b = 1` the `__destruct` method runs and modifies the value of `$b` to `10`, thus changing the result of the addition. There are a number of other ways in which a similar result can be achieved, for example by initializing the scope with references (causing aliasing) and by using error handler callbacks to modify variables at unexpected program points.

Because of this high degree of unpredictability we do not attempt to optimize pseudo-main code.

2.7. $this binding in methods

PHP's object orientation model draws many of its features from Java. As such it is relatively strict when compared to other dynamic languages, e.g. it does not allow replacing class methods at runtime ("monkey patching"). However, there are a number of type-safety loop-holes.

Instance methods in PHP have a `$this` object and a scope `self`, which refers to the class the method was declared in. One might expect the invariant `$this instanceof self` to hold, i.e. the `$this` object to be an instance of the class the method is being called on, or one of its children. However, this is not the case:

Prior to PHP 7.1 it was possible to force `$this` to refer to an arbitrary object, by using a combination of the reflection system and closure rebinding:

```php
class A {
    public function method() {
        var_dump(get_class($this)); // string(8) "stdClass"
    }
}
$closure = (new ReflectionMethod('A', 'method'))->getClosure(new A);
$closure->bindTo(new stdClass, 'A')();
```

This example acquires a reference to the method as a closure (anonymous function) and then calls it with `$this` bound to an `stdClass` instance (which is not an instance of `A`). This is problematic for optimization, because it makes it impossible to determine which method or property a `$this` access refers to, which is important for the purposes of type inference and specialization. As part of this work, we have effected this type of incompatible scope binding to be forbidden in PHP 7.1 [26], as such it is no longer a concern.

A less significant issue is that `$this` can also be entirely undefined inside an instance method. This occurs if an instance method is called statically, a legacy behavior that is still supported from PHP 4 days:

```
class A {
    public function method() {
        var_dump($this); // Error: Using $this when not in object context
    }
}
A::method(); // Deprecated: Non-static method A::method()
             // should not be called statically
```

However, this is less problematic, because any access to `$this` in this situation will generate an exception. As this causes the current control path to be left, we can safelty optimize as if `$this` were defined. The only optimization that requires special consideration is inlining, because we need to ensure that the exception is not elided.

2.8. Type annotations

PHP supports annotating function and method signatures with types:

```
function array_map(callable $fn, array $array) { /* ... */ }
```

Prior to PHP 7.0 it was only possible to annotate parameters and types were restricted to class/interface names, as well as `array` and `callable`. In PHP 7.0 support was expanded to allow scalar types and return type annotations:

```
function add(int $a, int $b): int {
    return $a + $b;
}
```

Unlike a number of other dynamic languages (like Hack or TypeScript) where type annotations are only used for verification by static analysis, type annotations in PHP are enforced by the runtime.[4] However, types are only checked on entry into and return from the function, they do not enforce that the type of a variable does not change. As such the following code is valid and will not generate a runtime error:

```
function foo(int $bar) {
    $bar = "string";
}
```

[4]Prior to PHP 7.0 it was possible to circumvent type checks by discarding recoverable errors in an error handler. In PHP 7.0 this loophole has been fixed as part of a move to exceptions.

Nonetheless these type annotations provide a valuable starting point for type-inference. However, as support for scalar types, which are the most valuable from an optimization perspective, has only been added recently, there is little code in the wild making use of them yet.

For PHP 7.1 a proposal has been submitted to add support for typed properties [32], which would guarantee that reads from such properties always returns a certain type:

```
class User {
    public int $id;
    public string $name;
}
```

This would significantly increase the amount of statically available type information, as it currently isn't feasible to determine that a property holds a specific type, e.g. due to the issues described in section 2.5. However, the typed properties proposal has been declined due to uncertainties about their semantics, so we cannot take advantage of this information (yet).

3. Prerequisites

This chapter describes a number of prerequisites for the analysis and optimization passes discussed in the next chapter. First, a very brief overview of PHP's compilation pipeline and instruction format is given. Then, we will proceed to introduce various structures needed for optimization, such as control flow graphs, dominator trees and most importantly static single assignment (SSA) form. As we're heavily dependent on SSA form, it is described in some detail and a number of PHP specific aspects are covered.

3.1. Compilation and execution pipeline

The PHP compiler is a classical compiler frontend, which takes input code, tokenizes it, builds an abstract syntax tree and finally compiles it to an intermediate representation (IR), which we call *opcodes*. At this point there are two ways to proceed, as illustrated in Figure 3.1:

The first is to directly execute the compiled opcodes on the virtual machine (VM). This implies that the opcodes for the entire application need to be recompiled on each web request (or other invocation of PHP). This deployment model is sometimes used by cheap shared hosting platforms, where many websites are hosted on a single server, with no particular expectation of performance.

The second model, which is used by professional PHP deployments, is to cache the compiled opcodes in a shared memory (SHM) opcode cache, so that compilation occurs only once. As in this case compilation performance is less important, some additional opcode optimizations can be applied.

The currently existing optimization pipeline is relatively simplistic: It performs a number of peephole optimizations and additionally also compacts the literals table and the temporary variable space. However, it does not perform any global data-flow optimizations. It is the goal of this work to extend the existing optimizer to perform such optimizations on static single assignment form.

It is important to realize that, while there are distinct compilation and execution phases, they are interleaved. Additional code is included at run-time, at which point opcodes for it are either compiled or loaded from SHM. At the time a file is compiled, it may be referencing functions and classes that have not yet been loaded.

The opcache extension, which both implements the SHM cache and hosts the optimizer, has an additional severe limitation: Not only does it have no knowledge of symbols that will be loaded in the future, it also has no knowledge of symbols loaded in the past. Each file is compiled completely independently, starting from a clean slate. This implies that

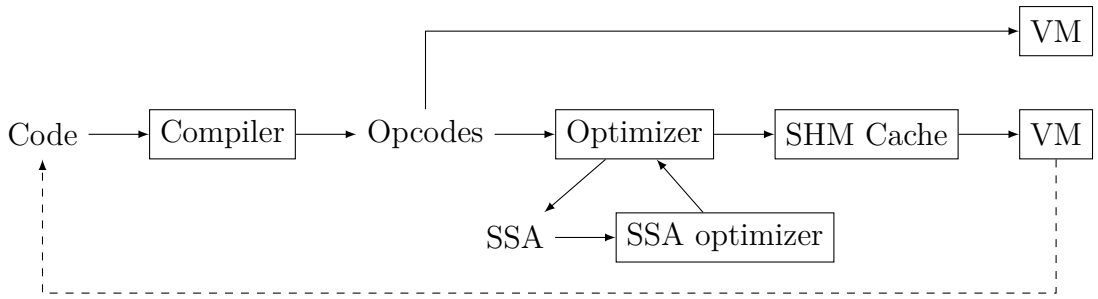


Figure 3.1.: PHP compilation and execution pipeline. First row: Opcodes are directly run on VM, recompiled on every request. Second row: Opcodes are compiled once and cached in SHM, minor optimizations are applied. Third row: The optimizer is extended to perform data-flow optimizations on SSA form. The dashed arrow represents that additional code can be loaded and compiled at run-time.

during optimization we do not know the signatures of any functions defined outside the current file, which requires the use of some very pessimistic assumptions.

We expect that we could achieve significantly better results if this limitation were removed. Even more advantageous would be a compilation mode similar to the RepoAuthoritative mode supported by HHVM, which requires all code to be known in advance and forbids run-time loading of additional code.

3.2. Instruction format

The PHP virtual machine uses what is essentially a three-address code, with each instruction having at most two input operands and one result operand. In some cases input operands are also written in-place.

There are three primary operand types: Constants, temporary variables and real variables. Constants can be not only integers and other scalar values, but also strings or arrays. They are stored in a separate literal table, rather than being encoded as instruction immediates.

Each variable used by a function is assigned a certain slot on the virtual machine stack. There are two (primary) types of variables: "Real" variables, which we also call compiled variables, correspond to actual variables in the program code, such as `$x`. Temporary variables on the other hand are introduced by the compiler, e.g. to hold the result of an expression.

There are several important differences in how these two variable types are handled: Real variables are fully initialized when a function is entered and destroyed when it is left. Instructions referencing real variables do not consume the variables, i.e. it is possible to use the same variable in multiple instructions. Temporary variables on the other hand 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 the value. As such, temporary variables can be read only once (unless the are assigned again).

We will not discuss the instruction format or the functioning of the virtual machine in further detail, as it is not particularly important for further considerations. However, the distinction between temporary and real variables is significant in a number of places.

3.3. Control flow graph

The PHP compiler emits instructions directly into a linearized array structure, with jump instructions using relative offsets. For global optimizations it is expedient to convert this linear structure into a *control flow graph* (CFG).

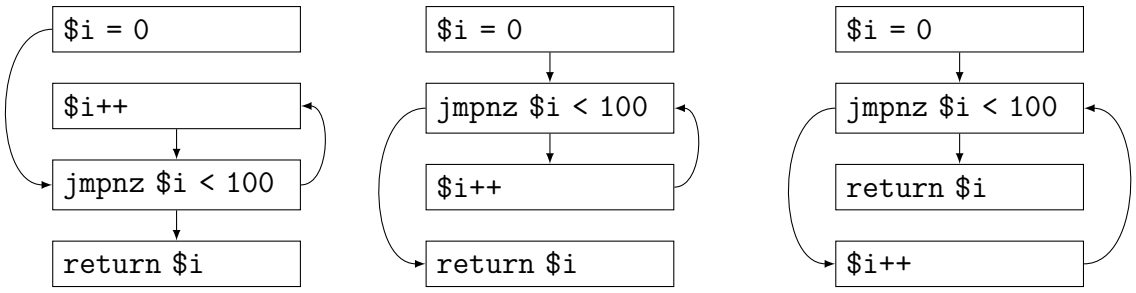
The vertices of a CFG are formed by *basic blocks*, which are linear instruction sequences with one entry point at the start of the block and one exit point at its end. Apart from these, there may be no jumps or jump targets inside a basic block. Edges in the CFG describe possible targets of a jump.

The block which is executed first upon entry into a function is designated the *entry* block. It is common to also designate a single dedicated *exit* block, through which control passes when leaving the function. We do not use a single exit block, instead any block that does not have outgoing edges is considered an exit block.

Many algorithms either require that the control flow graph be traversed in a certain order, or, even in cases where using a certain order is not necessary for correctness, will converge faster if the basic block ordering is chosen advantageously. When performing a depth-first search (DFS) of the CFG starting at the entry block, *preorder* is the order in which blocks are first entered, while *postorder* is the order in which they are left (after traversing all successors). Further, we say an edge (n, m) is a *back edge* if m is an ancestor of n in the DFS tree.

Postorder has the important property that, back edges notwithstanding, all successors are visited before their predecessors. Conversely, for reverse postorder (RPO) predecessors are visited before successors. As such RPO provides the most intuitive ordering of blocks, as it typically corresponds most closely to the original structure of the code. Both postorder and RPO are important for data-flow algorithms, as they allow propagation of data-flow properties (backwards in the case of postorder, forwards in the case of RPO) on non-cyclic graphs in a single iteration.

Further, we refer to the order in which the compiler has originally emitted the blocks as the *natural order*. In most cases this coincides with RPO, however loop conditions may occur out of order (to avoid an additional jump on every iteration). Figure 3.2 illustrates the different CFG orders for a simple incrementing for loop. Natural order and RPO here differ by the placement of the loop condition block. This example also illustrates that preorder and reverse postorder are not the same. The shown preorder is one of two possibilities, depending on the order in which successors of the loop condition block are visited (the other preorder is the same as RPO).



(a) Natural order (b) Reverse postorder (RPO) (c) Possible preorder

Figure 3.2.: Control flow graph orderings: (a) Natural order emitted by the compiler. (b) Reverse postorder, where, back edges notwithstanding, all predecessors occur before successors. (c) One possible preorder. The other is equivalent to RPO in this instance.

3.4. Dominance, dominator trees and dominance frontiers

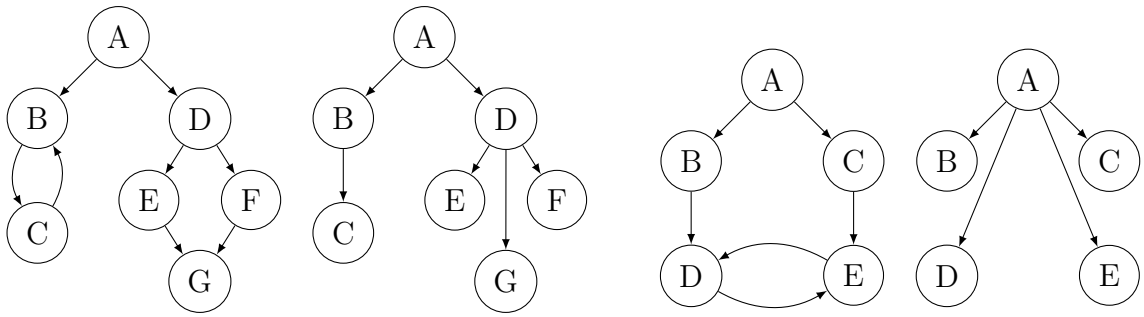
An important concept for optimization in general and static single assignment form in particular is that of dominance. A block n is said to *dominate* block m, if every path from the entry block to m contains n. n is said to *strictly dominate* m if additionally $n \neq m$ holds. In practical terms, this means that n always executes before m (if m executes at all).

The *immediate dominator* of n, idom(n), is the unique block which strictly dominates n, but does not strictly dominate any other block strictly dominating n (in other words, it is the "closest" strict dominator of n). The entry block has no immediate dominator, as it isn't strictly dominated by any block. The immediate domination relation imposes a tree structure on the CFG: This *dominator tree* is rooted at the entry block, and each node has as children the nodes it immediately dominates. This forms a tree structure because immediate dominators (parent nodes in the dominator tree) are unique.

Figure 3.3 shows dominator trees for two sample control flow graphs. Case (a) demonstrates the structure of the dominator tree for two common constructs, loops and if-then-else branches. Case (b) is presented to illustrate that the dominator tree is not always as obvious as in the former case. Here a CFG with an irreducible loop produces the degenerate case of a completely flat dominator tree.

Many algorithms with different complexity of implementation and asymptotic runtime are known for the construction of dominator trees. One of the most important ones is the Lengauer-Tarjan algorithm [24], whose asymptotic complexity is nearly linear in the size of the graph. We do not implement this algorithm and instead use the iterative data-flow algorithm by Cooper et al. [16], which has worse asymptotic complexity, but is significantly simper to implement and known to perform well for the relatively small graphs likely to be encountered by a compiler.[1]

[1]The paper by Cooper et al. claims that their algorithm outperforms Lengauer-Tarjan for compiler-



(a) Loop and If-Then-Else (b) Degenerate case

Figure 3.3.: Two control flow graphs and their corresponding dominator trees. Example (a) illustrates the structure of the dominator tree for two common syntax elements, namely loops and if-then-else structures. Example (b) shows a degenerate case involving an irreducible loop that results in a flat dominator tree.

Cooper's algorithm improves on an earlier algorithm attributed to Allen and Cocke [3], which iteratively solves the data-flow equations

$$\begin{aligned} \mathrm{Dom}(n_0) &= \{n_0\} \\ \mathrm{Dom}(n) &= \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p) \right) \cup \{n\}, \end{aligned}$$

where $\mathrm{Dom}(n)$ refers to the set of all dominators of node n, $\mathrm{preds}(n)$ the set of its predecessors, and n_0 the entry block. Cooper's "engineered" algorithm (Figure 3 in [16]) improves on this by implementing the dominator set intersection in a way that does not require maintaining the sets in memory and instead directly computes the dominator tree.

Another related concept that is important for SSA construction is that of *dominance frontiers*. The dominance frontier $\mathrm{DF}(n)$ of a node n is the set of all nodes m, for which n dominates an (immediate) predecessor of m, but does not strictly dominate m itself. Intuitively, the dominance frontier is where the dominance of n stops.

To compute dominance frontiers we use the algorithm in Figure 5 of the same paper [16]. This algorithm does not directly compute the dominance frontier of a node n, instead it computes the set of nodes which contain n in their dominance frontier.[2] This is accomplished simply by walking the dominator tree upwards, starting from the predecessors of a multiple-predecessor block n and ending at the immediate dominator of n, and adding all visited nodes (apart from the immediate dominator) to a set.

generated graphs. However this is disputed by Georgiadis et al. [21], who claim that the simple variant of Lengauer-Tarjan shows better performance characteristics even for small graphs, but do acknowledge that Cooper's algorithm is competitive.

[2] While this can obviously also be used to compute the dominance frontier sets proper, it is not necessary for the SSA construction approach we use.

3.5. Live-variable analysis

The construction of pruned SSA form (defined later in section 3.6.2), requires information about the liveness of variables. A variable is *live* at a certain program point, if a use of this variable is reachable through a path that does not include writes to this variable. Due to the assignment semantics discussed in section 2.3, in PHP we have to treat all assignments to real variables as both a read and write point. However, normal read/write semantics apply to VM temporaries.

We compute liveness information using classic iterative data-flow analysis: The set $\mathrm{Use}(n)$ is defined to contain all variables that are read before being written to in block n. The set $\mathrm{Def}(n)$ contains all variables that are written to in block n. Based on these sets, we consider the data-flow equations

$$\mathrm{Out}(n) = \bigcup_{s \in \mathrm{succs}(n)} \mathrm{In}(s)$$
$$\mathrm{In}(n) = \mathrm{Use}(n) \cup (\mathrm{Out}(n) \setminus \mathrm{Def}(n)),$$

where the $\mathrm{In}(n)$ and $\mathrm{Out}(n)$ set consist of the variables that are live at the start or end of block n, respectively. These equations can be solved by initializing $\mathrm{In}(n)$ and $\mathrm{Out}(n)$ to empty sets and iterating the equations until a fixed-point is reached.

As live-variable analysis propagates flow information backwards, postorder iteration is used to improve convergence speed. This ensures that, loops notwithstanding, successors will be visited before predecessors. To avoid recomputing sets that cannot have changed, a worklist is used, into which only predecessors of modified blocks are inserted. Lastly, all sets used in this computation are represented as bitsets of size #(blocks) × #(vars).

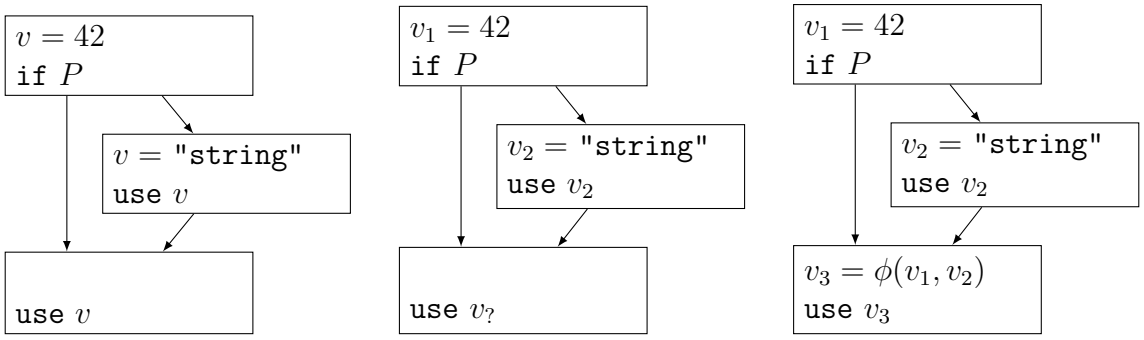
It should be noted, that the definition of the $\mathrm{Def}(n)$ set is flexible in that it can either include all variables written a block, or only those that are written before first use, thus making it symmetric with $\mathrm{Use}(n)$. Due to the structure of the flow equations, both definitions result in the same fixed-point. We use the former variant, because this form of the $\mathrm{Def}(n)$ set can later be reused during SSA construction.

3.6. Static single assignment form

As most of the analysis and optimization passes we have implemented operate on static single assignment form, this topic will be covered in some detail. First, the reasons for using SSA will be motived, then some of its properties discussed and the used SSA construction algorithm outlined. Furthermore we will discuss some specifics of SSA form when applied to PHP, and describe an extension of SSA form using π-nodes, which are used to improve the flow-sensitivity of value range and type inference. Our discussion of SSA properties is based on definitions from the SSA book [11].

3.6.1. Motivation

Before moving on to the more technical aspects of SSA construction, we would like to motivate why SSA form is commonly employed by optimizing compilers in general, and



(a) Non-SSA form (b) SSA precursor (c) SSA form

Figure 3.4.: Motivation for SSA form: In non-SSA code (a) modeling variable types requires associating the type with (variable, program point) pairs, while in SSA-form (c) the type can be associated directly with the variables. Figure (b) shows an intermediate consideration without ϕ nodes.

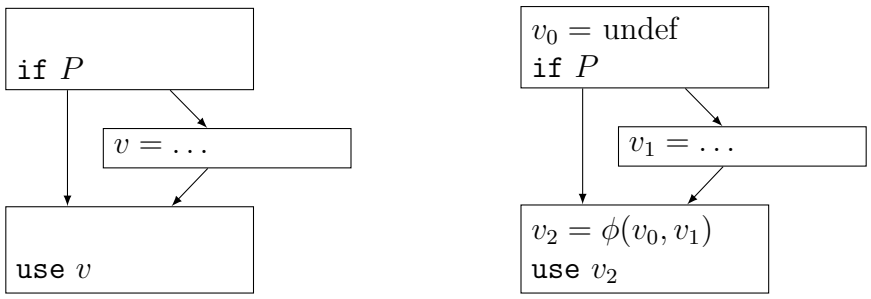
also by this project in particular. To this purpose we consider the example control flow graph shown in Figure 3.4 (a). In this example the variable v is written twice, once with an integer value and once with a string value. The variable is also used twice.

One question that is important for many analysis passes is what types certain variables may have at runtime. With the code in its current form, the best we can say about the variable v in general is that it may contain either an integer or a string. However, this information is not sufficiently precise, as it can be clearly seen that at the first use of v the variable always holds a string. To model this fact accurately, it is not sufficient to associate a type with a variable, instead it needs to be associated with a (variable, program point) pair. This representation is not memory efficient and generally hard to work with.

SSA form resolves this issue by splitting the variable v into multiple variables, one for each assignment, as shown in Figure 3.4 (b). This means that the integer value is assigned to v_1, while the string value is assigned to v_2. Uses of v are also renamed depending on which assignment reaches them.

The only problem with this scheme is that for the last use of v we cannot statically determine whether value v_1 or v_2 will reach it, as this depends on runtime control flow. SSA form solves this problem by introducing the concept of ϕ-nodes, as shown in Figure 3.4 (c). The statement $v_3 = \phi(v_1, v_2)$ creates a new variable v_3 that will either take the value of v_1 if the if-branch is not taken, or v_2 if it is taken.

Once the code is in SSA form, the original type analysis problem becomes much simpler: Now it is possible to associate type information directly with a variable, without any loss in accuracy. In our example v_1 will always be an integer, v_2 will always be a string and v_3 may be one of the two. Because each variable is only ever assigned once, we do not need to worry about the type (or value) of a variable changing from one program point to another.



(a) Non-strict program (b) Strict SSA form

Figure 3.5.: Figure (a) shows a non-strict program where v_1 may be used without being previously defined. Figure (b) illustrates how this program may be converted to strict SSA form by initializing the v variable to *undef* in the entry block. In strict SSA form a use is always dominated by the definition.

3.6.2. SSA properties: Minimal, pruned, strict

The main property of static single assignment form, from which it also derives its name, is that each SSA variable is only assigned once. SSA construction is usually performed in a two-step process, where first ϕ-nodes are placed as necessary, and then variables are renamed to use a unique name for each assignment. In order for the second step to be possible, each use of a variable must have a unique *reaching definition*, where a definition D of variable v is said to *reach* a program point p if there exists a path from the definition D to p that does not contain another definition of v. This *unique reaching definition property* is what needs to be established by ϕ-node placement.

While this is the only requirement that SSA form must satisfy, there are a number of additional properties that may be desirable. Firstly, we would like to only place as many ϕ-nodes as necessary, to reduce the number of SSA variables and the size of the SSA graph. There are two different interpretations for whether a ϕ-node is "necessary":

Minimal SSA form requires that the minimum number of ϕ-nodes is placed, which still ensures that there is a unique reaching definition at *every* program point. In particular this means that ϕ-nodes will be placed even if the variable has no subsequent uses. *Pruned SSA form* on the other hand requires a minimum number of ϕ-nodes, such that every *use* has a unique reaching definition. Compared to minimal SSA form, this means that ϕ-nodes will only be placed if the variable is also live-in at the block.

Another possibility is the use of *semi-pruned* SSA, which may place ϕ-nodes for variables that are not live-in, but only does so if the variable is not block-local, i.e. there is a basic block in which the variable is used before defined. However, as we treat assignments as both uses and definitions, all non-temporary variables would be non-local, degrading the usefulness of semi-pruned SSA form. For this reason the PHP implementation chose to use pruned SSA form, which is why the iterative liveness analysis pass is performed prior to SSA construction.

A further property of the SSA graph that we require is *strictness*, which means that variables are always defined before they are used, for any control flow path starting at the

entry block. In PHP this property is not automatically given, because it is possible to use variables before they have been initialized. Figure 3.5 (a) shows an example CFG where a variable is only initialized if a predicate P holds, but the variable is always used. This program satisfies SSA form as defined here, as the use of v is reached only by a single definition.

To ensure that the SSA form is strict, we insert a virtual (not materialized in the instructions) assignment to an *undef* value for each variable that could potentially be used before initialization (i.e. for all variables that are not VM temporaries). This is illustrated in Figure 3.5 (b), however we will not include these virtual definitions from now on.

As there is always a unique reaching definition, strictness in SSA form reduces to a simpler condition: An SSA form program is strict, if each use of a variable is dominated by its definition. This *dominance property* is important for the correctness of many of the algorithms operating on SSA.

3.6.3. Construction of SSA form

For construction of SSA form PHP follows the general approach by Cytron et al. [17], although some of the specifics differ. Cytron's SSA construction algorithm proceeds in two steps: First, ϕ-nodes are placed to ensure the unique reaching definition property. Subsequently variables are renamed such that each definition uses a unique name.

To determine where placement of ϕ-nodes is necessary, it is useful to consider the concept of join sets. A node n is a *join node* of nodes $a \neq b$ if there exist non-empty paths from a to n and b to n such that the paths (apart from node n) are disjoint. The *join set* $J(S)$ contains the pairwise join nodes of set S.

A minimal placement of ϕ-nodes is then intuitively given by the set $J(D_v)$ where D_v denotes the blocks containing definitions of v, as $J(D_v)$ accurately captures the nodes where two different definitions of v join.[3] However, to ensure that the resulting SSA form is strict, the implicit definition in the entry block e must be accounted for, so that the actual set of ϕ placements is $J(D_v \cup \{e\})$.

A primary result of Cytron's work is that this set is equivalent to the *iterated dominance frontier* $DF^+(D_v)$ of D_v, where $DF^+(S)$ may be defined as the fixed point of the recurrence relation

$$\begin{aligned} DF_1(S) &:= DF(S) \\ DF_{i+1}(S) &:= DF(S \cup DF_i(S)), \end{aligned}$$

with $DF(S)$ denoting the union of the dominance frontiers of all elements in S. Intuitively the correspondence between $DF^+(D_v)$ and $J(D_v \cup \{e\})$ exists, because $DF(D_v)$ captures the nodes where the dominance of a definition of v ends and as such joins with other definitions of v (or at least with the implicit definition in the entry block). The reason

[3] Wolfe [35] showed that $J(S \cup J(S)) = J(S)$, for this reason it is sufficient to consider $J(S)$ directly, rather than an iterated join set $J^+(S)$, as is necessary for dominance frontiers.

why the dominance frontier needs to be iterated is that each placed ϕ-node also constitutes a definition of the variable, and may thus require placement of further ϕ-nodes.

We perform ϕ-node placement in two steps: During the iterative live-variable analysis the $\mathrm{Def}(n)$ and $\mathrm{In}(n)$ sets were computed, which contain the variables that are, respectively, defined in or live-in at block n. Based on these sets we iterate the equations

$$\mathrm{Phi}(n) = \mathrm{In}(n) \cap \bigcup_{m \in DF^{-1}(n)} \mathrm{Def}(m)$$
$$\mathrm{Def}(n) = \mathrm{Def}(n) \cup \mathrm{Phi}(n)$$

to a fixed point, with $\mathrm{Phi}(n)$ being the set of variables for which a ϕ-node must be placed in block n and $\mathrm{DF}^{-1}(n)$ the set of nodes in whose dominance frontier n is contained, i.e. precisely the set computed by Cooper's dominance frontier algorithm described in section 3.4. The computation of the $\mathrm{Phi}(n)$ set intersects with $\mathrm{In}(n)$, because we are using pruned SSA form and thus only place ϕ-nodes for live variables. In the second step ϕ-nodes are placed in a single pass based on the computed $\mathrm{Phi}(n)$ sets.

This differs slightly from the approach described by Cytron et al., which explicitly computes the $DF(n)$ sets and then, for each variable individually, uses a block worklist to handle dominance frontier iteration, while keeping track of which ϕ-nodes have already been placed.

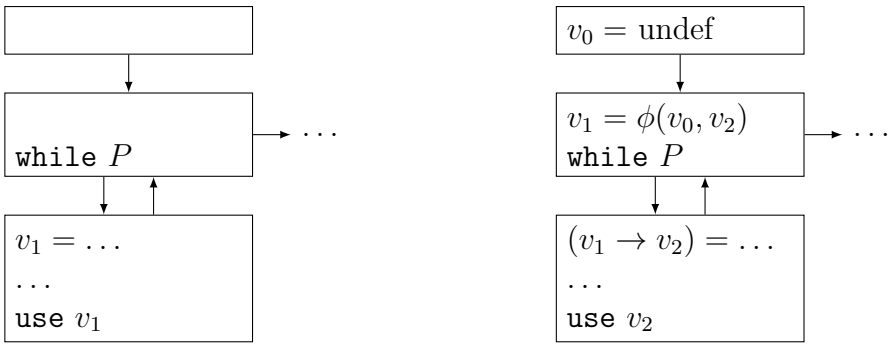
The second part of SSA construction is variable renaming, where each definition of a variable is assigned a unique name and uses are adjusted accordingly. We follow the general approach of Cytron et al., which works by performing a preorder walk of the dominator tree, while keeping a name stack for every variable. When encountering a use of a variable, it will be replaced with the top name from the stack. When encountering a definition, a newly generated name is pushed on the respective stack. After a node and all its (dominator tree) children have been processed, the name stacks are restored to their previous state. ϕ-nodes require slightly special handling, in that a ϕ argument corresponding to predecessor p must be renamed after block p, but not its children, has been processed (rather than simply handling ϕ-nodes as ordinary instructions at the start of the block).

Cytron suggests the use of separate index spaces for each original variable, e.g. numbering variables v_1, v_2, v_3 and w_1, w_2, w_3. We use a shared index space instead, i.e. number v_1, v_2, v_3, w_4, w_5, w_6. While this is more useful for internal handling, in examples we'll continue to number each variable separately. We further deviate from Cytron's construction algorithm, in that we will directly replace the top element of the name stack if a variable is assigned multiple times in a basic block, rather than pushing a new entry to the stack each time. This simple improvement is described in [10].

3.6.4. Specifics of SSA form in PHP

As discussed in section 2.3, assignments in PHP must be treated as both a use and definition point of the variable.[4] While this poses no fundamental problem to the SSA

[4]Based on type information inferred at a later point, it would be possible to remove these additional uses, if the type is known to not be affected by the issues from section 2.3. However, we have found



(a) Ordinary SSA form

(b) SSA form with assignments treated as both uses and definitions

Figure 3.6.: To account for certain semantics of the PHP language, assignments are treated both as use and definition points. Subfigure (a) shows ordinary SSA form for a loop body local variable, while (b) illustrates the necessary changes if assignments are treated as uses.

paradigm, it does significantly change the structure of the SSA graph.

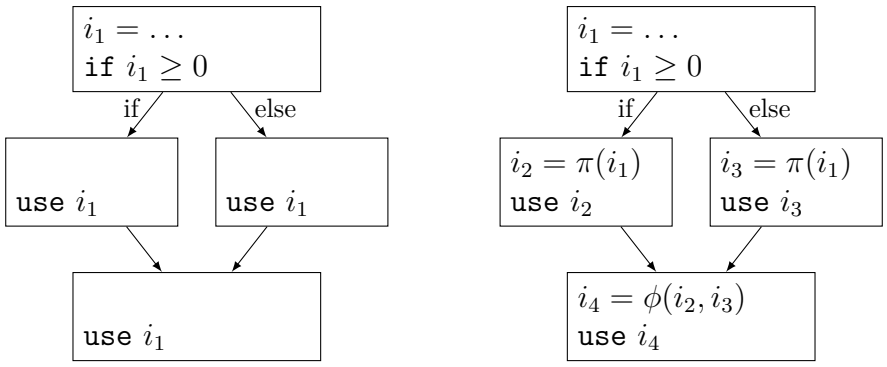
Figure 3.6 (a) shows the control flow graph of a simple while loop, in whose body the variable v_1 is used only locally, i.e. v_1 is defined and used in the same block, and not anywhere else. In this case (pruned) SSA form does not require placement of ϕ-nodes, because v_1 is not live-in at any block.

In PHP the situation changes: The assignment of v_1 is also considered a use of the variable, as such it is live-in both at the loop header and body. This requires the placement of a ϕ-node in the loop header, as shown in Figure 3.6 (b). The notation $(v_1 \to v_2) = \ldots$ is used to signify that the assignment uses the old value v_1 and generates the new value v_2. We refer to such uses as *improper uses* and commonly need to treat them specially in analysis passes. In the interest of clarity, figures will continue to treat assignments as definitions only, unless the distinction is important.

It is common that SSA implementations desugar operations like $a \mathrel{+}= 1$ into $a = a + 1$, such that the number of different instructions is reduced and the instruction format is more uniform. We do not perform such a transformation and instead support SSA instructions that perform in-place modifications. Using the same notation as for ordinary assignments $(a_1 \to a_2) \mathrel{+}= 1$ performs the equivalent of an $a_2 = a_1 + 1$ operation. The reason why this approach is chosen, apart from a desire to match the existing instruction set as closely as possible, is that subtleties of behavior may be lost in such a transformation. For example the `++$a` operation in PHP is not equivalent to `$a = $a + 1`, because string increment and string addition behave differently, but there are a number of more subtle discrepancies as well, particularly when the left-hand-side of the compound assignment is an array offset or object property.

A few words should be said about the in-memory representation of the SSA graph: The

that even in this case it is useful to retain them in order to maintain the conventionality property discussed in section 4.7.



(a) Normal SSA form (b) Extended SSA with π-nodes

Figure 3.7.: Motivational example for extended SSA form: (a) shows a CFG where information about the value range of i_1 is only available in certain branches, but the same variable is used everywhere. (b) solves this by splitting the variable using π-nodes.

SSA implementation used by PHP generally tries to maintain a very close correspondence to the original non-SSA program. As such, the SSA information is implemented as an overlay over the non-SSA instruction stream. Each of the original non-SSA instructions has (up to) three operands: op1, op2 and result. The SSA overlay then specifies the SSA variable number that is used or defined by each of the three operands: op1_use, op1_def, op2_use, op2_def, result_use and result_def. It is possible for one operand to make use of both the use and def fields, e.g. for $(a_1 \to a_2) = 1$ op1_use is a_1 and op1_def is a_2.

ϕ-nodes are stored separately as a linked list associated with the basic blocks of the CFG. Information about the variables that particular SSA numbers refer to (such as which of the non-SSA variables they correspond to) is kept in a separate array. Variables also store their unique defining instruction and a linked list of use-sites (def-use chain).

3.6.5. Extended SSA form: Pi nodes

One advantage of SSA form is that it makes it relatively easy to lend a degree of (control) flow-sensitivity to algorithms, a topic which will be discussed more closely in section 4.1. However, there are cases where the variable splitting performed by SSA construction is not sufficient to capture some control-sensitive properties.

Such a case is illustrated in Figure 3.7 (a), where a variable i_1 is used in three separate blocks. Ideally, we should be able to make use of the fact that the use in the if-branch is necessarily non-negative, while the use in the else-branch is always negative. However both uses refer to the same variable i_1, so tracking this information would require returning to an approach where data-flow properties depend not only on the used variable, but also on the location of the use.

This problem can be solved by artificially splitting variables using π-nodes, as is shown in Figure 3.7 (b). A π-node is placed at the start of both branches, thus creating separate variables i_2 and i_3, with which the range information can be associated. An additional ϕ-

node needs to be placed at the join point of the two branches, where the range information associated with i_2 and i_3 will cancel out.

The concept of π-nodes was adopted from the ABCD array bounds check elimination algorithm [6], which refers to the resulting SSA form as "extended SSA" or e-SSA. As the paper provides little information on how to perform π-placement, we will cover this in more detail in the next section.

At least in our implementation, π-nodes are used not only to split variable ranges, but also to store information about the associated condition. For this reason we may place π-nodes even in blocks that already have a ϕ-node for the same variable: While there is no need to split the variable in this case, we still need the π-node to store condition information.

At present π-nodes are used in value range inference and type inference. π-nodes are typically placed in pairs, with one node π_t in the true-branch of the condition and another π_f in the false-branch, both carrying complementary range or type constraints. We recognize the following general classes of conditions and associated constraints:

1. Simple range constraints: A condition $v > N$ will result in two π nodes with the following constraints, where ∞ denotes a potential integer overflow:

$$\pi_t(v) : [N + 1, \infty]$$
$$\pi_f(v) : [-\infty, N]$$

2. Symbolic range constraints: A condition $a + N > b + M$ will result in:

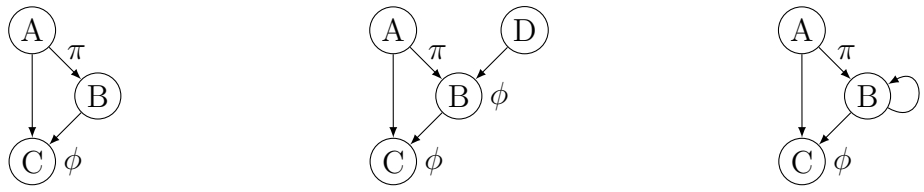
$$\pi_t(a) : [b + M - N + 1, \infty]$$
$$\pi_t(b) : [-\infty, a + N - M - 1]$$
$$\pi_f(a) : [-\infty, b + M - N]$$
$$\pi_f(b) : [a + N - M, \infty]$$

   Here the value range of variable a depends on the value range of variable b and vice versa. Each branch needs two π-nodes to capture the information for both variables.

3. Type constraints: A condition `is_int(`v`)` results in a π_t-node asserting that the variable is always an integer, and a π_f-node asserting that it may be anything but an integer.

The concept of π-nodes is very general and can be easily extended to cover additional conditions, such as `isset($array['key'])` checks to determine if array keys exist.

Placing π-nodes whenever it is possible may sometimes result in trivially useless π-nodes. For example, if for the CFG from Figure 3.7 the else-block were removed, placing a π-node at the target of the else-edge (now the last block) would be useless, as this SSA variable would be immediately used in a ϕ-node that annihilates the constraints of a positive and a negative π-node. However, the π-node might still be useful if the if-block contained an assignment to i. As we haven't found any *simple* heuristic for when π-nodes will be trivially useless, we prefer always placing them (for live variables).



(a) Target has single predecessor (b) Target has multiple predecessors (c) Target has multiple dominated predecessors

Figure 3.8.: Control flow graphs illustrating minimal ϕ-placements after the addition of a π-node, which is logically located on a CFG edge. ϕ-placement depends on number and domination relation of predecessors of the target block B.

3.6.6. Phi placement after pi placement

While for the purposes of presentation (and storage) it is useful to place π-nodes at the start of basic blocks, they are semantically located along control flow edges. For this reason placement of ϕ-nodes to account for new π-definitions requires special care, as the standard approaches assume that definitions occur only within basic blocks, rather than along control flow edges. Figure 3.8 shows a number of control flow graphs with π-nodes and corresponding minimal ϕ-placements. The target of the edge with the π-node is always block B. The different cases will be discussed in the following.

The simplest case is illustrated in subfigure (a), where the target B only has a single predecessor. We can treat this instance as if the π-definition were located at the start of the B block, and thus need to place ϕ-nodes on $\mathrm{DF}^+(B)$.

Subfigure (b) shows a case where the target B has multiple predecessors. Placing ϕ's only on $\mathrm{DF}^+(B)$ would be insufficient, as two different definitions of the variable flow into B. As such, ϕ-nodes must be placed on $\{B\} \cup \mathrm{DF}^+(B)$.

However, this rule does not lead to minimal SSA form, as is shown in subfigure (c). In this case the target B has multiple predecessors, but B dominates all predecessors (not counting the source of the π-edge), as such it is not necessary to place a ϕ-node in B. Only placing ϕ-nodes on $\mathrm{DF}^+(B)$ would also be non-minimal, as B is part of its own dominance frontier in this example. The minimal ϕ-placement is $\mathrm{DF}^+(\mathrm{DF}(B) \setminus \{B\})$.

To more rigorously show that these intuitive considerations are correct, we may split an edge (s, t) along which a π-node is located into two edges (s, p) and (p, t), with p being a newly inserted block containing the π-definition. The problem of ϕ-placement then reduces to finding the iterated dominance frontier $\mathrm{DF}^+(p)$. As an auxiliary statement, we claim that p dominates t iff t dominates all $\mathrm{preds}(t) \setminus \{p\}$.

Proof: ($\Rightarrow$) Assume p dominates t. As $p \neq t$ this implies that p dominates all $\mathrm{preds}(t)$. Thus all paths from the entry block to $\mathrm{preds}(t)$ must contain p. However, as p only has a single successor t, every such path that does not end in p also contains t. Thus t dominates all $\mathrm{preds}(t) \setminus \{p\}$. ($\Leftarrow$) Assume t dominates all $\mathrm{preds}(t) \setminus \{p\}$, i.e. there exists no path from entry to $\mathrm{preds}(t) \setminus \{p\}$ that does not contain t. However, as t is not the entry block, any such path must contain at least one $\mathrm{preds}(t)$, which can only be p. As such p dominates $\mathrm{preds}(t) \setminus \{p\}$. As p trivially dominates itself, p dominates all $\mathrm{preds}(t)$ and thus p also

dominates t. □

We now wish to establish a relationship between $\mathrm{DF}^+(p)$ and $\mathrm{DF}^+(t)$. If p does not dominate (its only successor) t, then $\mathrm{DF}(p) = \{t\}$ and $\mathrm{DF}^+(p) = \{t\} \cup \mathrm{DF}^+(t)$. The case where p dominates t is slightly more complicated: Let $\mathrm{dom}(p)$ refer to the nodes dominated by p and $\mathrm{sdom}(p)$ to those strictly dominated by p. As domination is transitive and t is the only successor of p, it holds that $\mathrm{dom}(p) = \mathrm{dom}(t) \cup \{p\}$ and $\mathrm{sdom}(p) = \mathrm{sdom}(t) \cup \{t\}$. As such, the dominance frontier of p is given by

$$\begin{aligned}\mathrm{DF}(p) &= \{n \mid \exists m \in \mathrm{preds}(n) : m \in \mathrm{dom}(p) \wedge n \notin \mathrm{sdom}(p)\} \\ &= \{n \mid \exists m \in \mathrm{preds}(n) : (m \in \mathrm{dom}(t) \vee m = p) \wedge (n \notin \mathrm{sdom}(t) \wedge n \neq t)\},\end{aligned}$$

where $m = p$ is only possible for $n = t$, which is excluded by the later condition. Thus:

$$\begin{aligned}\mathrm{DF}(p) &= \{n \mid \exists m \in \mathrm{preds}(n) : m \in \mathrm{dom}(t) \wedge n \notin \mathrm{sdom}(t)\} \setminus \{t\} \\ &= \mathrm{DF}(t) \setminus \{t\}\end{aligned}$$

Consequently the iterated dominance frontier is $\mathrm{DF}^+(p) = \mathrm{DF}^+(DF(t) \setminus \{t\})$. Note that for the special case where t does not have any predecessors apart from p, $\mathrm{DF}(t)$ cannot contain t, as such this expression reduces to $\mathrm{DF}^+(p) = \mathrm{DF}^+(t)$. With this we have arrived at the same sets as previously discussed.

4. Analysis and Optimization

This chapter describes the various analysis and optimization passes that have been implemented and constitutes the main part of this thesis.

We begin by introducing a general flow-sensitive propagation framework for data-flow properties and then describe type inference and constant propagation based on it. Next, dead code elimination and type specialization are introduced. This is followed by a discussion of liveness oracles on SSA form and how to perform copy propagation on conventional SSA form using them. Subsequently function inlining and propagation of information using the dominator tree are discussed, both of which are not primary objectives of this work. Finally, we make a brief note on testing.

4.1. Sparse conditional propagation of data-flow properties

Many analysis passes are interested in associating some kind of data-flow property with SSA variables. Examples of possible properties are "variable has constant value C", "variable has type T" or "variable has same value as other variable v". This mapping should be as precise as possible, while being cheap to compute.

For many data-flow properties, this problem can be solved using the *sparse conditional constant propagation* (SCCP) algorithm due to Wegman and Zadeck [33]. While this algorithm is specific to constant propagation, it can be easily extended into a more general propagation framework. We will present such a generalization here, based on the description in the SSA book [9].

The SCCP algorithm has three desirable properties, which set it apart from classical non-SSA iterative data-flow algorithms. Firstly, it is *sparse*, meaning that a changing property of a variable can be directly propagated only to its use sites, rather than requiring that the flow information for entire basic blocks is recomputed. Secondly, the algorithm is *optimistic*, which means that it starts from an optimistic assumption like "all variables are constant" and may later disprove this assumption. Especially for code with loops, this leads to better result than starting with a pessimistic assumption. Thirdly, the algorithm is *conditional*, i.e. has at least some measure of (control) flow-sensitivity, in that it tracks which CFG edges are feasible under the current propagation results.

4.1.1. Requirements

The algorithm operates on a bounded lattice $(L, \sqsubseteq)$, which means that for every finite subset $S \subseteq L$ there exists a least upper bound $\bigsqcup S$ (join) and greatest lower bound $\bigsqcap S$

(meet) over the partial order $\sqsubseteq$. As this also applies to the empty set, there exists a least element $\bot := \bigsqcup \emptyset$ and greatest element $\top := \bigsqcap \emptyset$. We will formulate the general algorithm using meet operations, but for specific applications we may switch to a dual lattice, if the use of joins is more convenient.

Each SSA variable v is associated with a value $\mathit{Value}(v)$ from this lattice. The importance of the meet operation lies in the handling of ϕ-nodes: Ignoring flow-sensitivity, the value of the ϕ result variable is simply the meet of the values of the ϕ-operands. This is particularly intuitive for a type lattice, where the result type of a ϕ-node is the union of the possible types of the ϕ-operands. Here the set union acts as the meet (more conventionally: join) operator.

Next to the lattice, we require a transfer function $\mathrm{eval}(I, v)$ which computes a new value for variable v defined by instruction I based on the current state of the $\mathit{Value}(\cdot)$ relation. This function must be monotonic in the values of the SSA variables used by I. For example, given an instruction $I : v = a \text{ op } b$, the function must be monotonic in $\mathit{Value}(a)$ and $\mathit{Value}(b)$. For a type lattice, this would mean that a *more* precise knowledge of the possible types of the input operands of an instruction cannot make the possible output types, as determined by $\mathrm{eval}(\cdot)$, *less* precise.

Lastly, we require a feasible-successors(b) function, which returns a set of *feasible* successors of basic block b. A successor s of block b is feasible, if it is possible for the control flow edge (b, s) to be taken, given the current $\mathit{Value}(\cdot)$ relation. For example, if we are working on a constant propagation lattice and consider a branch on a variable with current value `true`, only one of the successors is feasible. If feasible-successors(b) is defined to return all successors of b, then the algorithm reduces to a non-conditional variant.

4.1.2. Algorithm

The fundamental idea of the unconditional data-flow propagation framework is simple: Start with an optimistic assumption of $\top$ for all variables and for each instruction either run $\mathrm{eval}(\cdot)$ or use the meet operation if it is a ϕ-node. If this lowers the lattice value, reevaluate all instructions using the value and iterate this until a fixed point is reached. The conditional framework extends this by also keeping track of which CFG edges are currently feasible, and ignoring those that aren't.

The main loop of the algorithm is shown in Listing 1. Two worklists *InstrWorklist* and *BlockWorklist* are used to keep track of instructions and blocks that still need to be processed. Additionally *FeasibleEdges* stores all CFG edges that are currently feasible and *ExecutableBlocks* contains blocks that are both executable, i.e. have at least one incoming feasible edge, and have been processed once already. The latter condition is only important insofar as it reduces the number of evaluations.

All sets are initialized to be empty, only the entry block is included in the *BlockWorklist*, as it is the only block for which we know a priori that it is executable. $\mathit{Value}(\cdot)$ is initialized to $\top$ for all variables that are not implicitly defined in the entry block (i.e. effectively "undefined" variables). The initialization of the implicit variables is lattice dependent.

The algorithm proceeds to pop elements from the *InstrWorklist* or *BlockWorklist* until both are empty. It does not matter for correctness in which order these worklists are

Listing 1 Main loop of propagation framework

```
InstrWorklist ← ∅
BlockWorklist ← {entry}
ExecutableBlocks ← ∅
FeasibleEdges ← ∅
Initialize Value(v) for all SSA variables v

while InstrWorklist ≠ ∅ ∨ BlockWorklist ≠ ∅ do
    while InstrWorklist ≠ ∅ do
        I ← any element of InstrWorklist
        InstrWorklist ← InstrWorklist \ {I}
        b ← basic block that contains I
        if b ∈ ExecutableBlocks then
            visitInstr(I)

    while BlockWorklist ≠ ∅ do
        b ← any element of BlockWorklist
        BlockWorklist ← BlockWorklist \ {b}
        ExecutableBlocks ← ExecutableBlocks ∪ {b}
        for all instructions I in block b do
            InstrWorklist ← InstrWorklist \ {I}
            visitInstr(I)
```

processed. If an instruction is popped and the basic block it is contained in is executable, the instruction is visited. If a block is popped, it is marked as executable and all its instructions are visited. While this happens the visited instructions should be removed from the instruction worklist, to avoid needless reevaluations.

The more interesting part of the algorithm is the handling of individual visited instructions as shown in Listing 2. We distinguish two cases: ϕ-instructions and everything else.

For a ϕ-instruction in block b with result variable v_0 and operands v_i from predecessor blocks p_i, we compute the new value of v_0 as

$$\mathit{Value}(v_0) = \bigsqcap\{\mathit{Value}(v_i) \mid (p_i, b) \in \mathit{FeasibleEdges}\},$$

i.e. we take the meet of all $\mathit{Value}(v_i)$ for which the edge (p_i, b) is feasible. As such, if a block can be entered along multiple edges, but only some of these edges are known to be executable, only values flowing along the executable edges are considered. If the newly computed $\mathit{Value}(v_0)$ differs from the previous, all instructions that use v_0 are added to the *InstrWorklist*, so they will be reevaluated with the new value.

For all other instructions we compute the new values for all variables v defined by the instruction I using $\mathrm{eval}(I, v)$. Once again, if the value of a variable changes, all uses of the variables are added to the instruction worklist. For both ϕ and non-ϕ instructions

Listing 2 Handling of individual instructions in the propagation framework

```
procedure VISITINSTR(instruction I)
    b ← basic block that contains I
    if I is v_0 = φ(v_1, ..., v_n) then
        newValue ← ⊤
        for all φ-operands v_i do
            p_i ← predecessor block for operand v_i
            if (p_i, b) ∈ FeasibleEdges then
                newValue ← newValue ⊓ Value(v_i)
        if newValue ≠ Value(v_0) then
            Value(v_0) ← newValue
            InstrWorklist ← InstrWorklist ∪ uses(v_0)
    else
        for all variables v defined by I do
            newValue ← eval(I, v)
            if newValue ≠ Value(v) then
                Value(v) ← newValue
                InstrWorklist ← InstrWorklist ∪ uses(v)
        if I terminates the basic block then
            for all s ∈ feasible-successors(b) do
                MARKEDGEFEASIBLE(b, s)
```

it may be advisable to check whether the current value is $\bot$, in which case reevaluation is not necessary: As we require all lattice transitions to be monotonic, a $\bot$ value cannot change to anything else.

Lastly, we need to consider the case where the visited instruction is a branch terminating the basic block b, in which case the values of its input operands may influence the feasible successors of the basic block. All edges (b, s) for $s \in$ feasible-successors(b) are marked feasible according to the procedure in Listing 3:

Assuming the edge is not already contained, it is added to the *FeasibleEdges* set. Then, if the target block is not yet in *ExecutableBlocks*, it is added to the *BlockWorklist* (and will be added to *ExecutableBlocks* when it is popped from the block worklist). If the target is already marked executable, then only the ϕ-instructions in the target block are revisited. This is necessary because the result value of the ϕ-node was computed using operand values from feasible edges only, so the value from the newly feasible edge has not been taken into account.

4.1.3. Properties

The height h of a lattice $(L, \sqsubseteq)$ is the maximum length of a chain $l_1 \sqsubseteq l_2 \sqsubseteq \cdots \sqsubseteq l_n$, such that $l_i \neq l_j$ for $i \neq j$. As lattice transitions in the propagation framework are always monotonic, the value of a variable may be lowered at most $h - 1$ times, causing

Listing 3 Marking edges as feasible in propagation framework

```
procedure markEdgeFeasible(source block s, target block t)
    if (s, t) ∉ FeasibleEdges then
        FeasibleEdges ← FeasibleEdges ∪ {(s, t)}
        if t ∈ ExecutableBlocks then
            for all φ-instructions I in t do
                visitInstr(I)
        else
            BlockWorklist ← BlockWorklist ∪ {t}
```

the reevaluation of its uses. Additionally, each control flow edge may become feasible at most once. If we denote the total number of uses of SSA variables as N (edges in the SSA graph) and the number of edges in the control flow graph as M, the upper bound on execution time is $O((h-1)N + M)$.

The algorithm propagates data-flow properties and detects unreachable code (as far as it can be detected using a given lattice) simultaneously. This is strictly more powerful than performing unconditional propagation and unreachable code elimination in sequence any number of times, because lattice values contributed from otherwise unreachable code paths could inhibit their being detected unreachable.

Not all data-flow problems that can be modeled using classical iterative data-flow analysis are supported by this propagation framework. In particular it is limited to forward-directed problems. This is a rather fundamental limitation of operating on SSA form, as ϕ-nodes are placed based on forward control flow. To support backwards-directed problems a different representation (such as static single information) is necessary. Furthermore analyses such as available expressions cannot be modeled, as propagation to use-sites of variables is not sufficient for them.

It should further be noted that the algorithm as introduced here only operates on individual procedures. It is possible to extend it to an inter-procedural variant that is able to propagate information across call boundaries. We did not attempt to implement an inter-procedural version due to time constraints, but this would be an obvious avenue for future improvement.

4.2. Type inference

Many of the optimizations that will be discussed in the following depend, in one way or another, on the availability of type information for variables. For type specialization the dependency is obvious, but many other passes also require some type-related knowledge. In particular, variables that may be references or may be undefined typically have to be excluded from transformations. Furthermore it may be useful to know whether a variable can hold a reference-counted value. Types can also help to determine whether or not an instruction may generate an error.

We implement type inference using the propagation framework described in section

4.1. As such, the term "type inference" is somewhat misleading in this context, and the procedure might be more accurately referred to as "type propagation". In particular, this approach works by starting with known type information (e.g. from variable initializations), propagating this information to use-sites, and determining which output types are possible for a set of input types. Notably, this implies that type information only flows forwards.

Type inference that also admits backwards flowing type information would not be useful for our purposes, as we do not have the option of restricting types of parameters and other sources. Even if we can determine that a parameter can only reasonably be an integer, the function must still be capable of handling other types. One way in which this could still be applicable, is the use of multi-versioned functions, which check at the start of the function whether the types of the given parameters coincide with the inferred types, and then either dispatch to a version based on the inferred type information, or to the original unconstrained version. However, we did not further pursue such an approach in this work.

The following subsections will describe how the propagation framework is used for type inference, in particular which lattice is used, how the transfer function is defined and how feasible successors are determined. We also discuss how π-nodes are employed to improve the inference quality and lastly introduce an additional type narrowing step used to avoid union types in certain situations.

4.2.1. Type lattice

The lattice $(\mathcal{T}, \sqsubseteq)$ used for type inference can, with a few exceptions outlined later, be approximated as a power set lattice $(\mathcal{P}(T), \subseteq)$ 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. To denote such type sets we will use union type notation, i.e. write *int*|*double* instead of {*int*, *double*}, and as a result won't make a strong distinction between an individual type and a set containing only that type.

The basic types supported by PHP are *null*, *bool*, *int*, *double*, *string*, *array*, *object* and *resource*. All of these types are part of the type universe T, however some of them are specialized further:

The *bool* type is subdivided into the pseudo-types *true* and *false*. This matches the underlying implementation (which also treats true and false as separate types, even though they are not at the language level) and the distinction is useful for flow-sensitive type inference. As such *bool* is the same as *true*|*false*.

The *object* type may be further restricted to objects of a specific class: *object*(`Foo`) refers to an object of class `Foo`, while *object*(*instanceof* `Foo`) also allows subclasses of `Foo`.[1] The ordering between the different object types is given by

$$\textit{object}(\texttt{Foo}) \sqsubseteq \textit{object}(\textit{instanceof}\ \texttt{Foo}) \sqsubseteq \textit{object}$$

[1]Unless it is contextually relevant, we will not strongly distinguish between classes and interfaces. As such *object*(*instanceof* `Foo`) can equally refer to an object implementing interface `Foo`.

and further, for a class `Bar` extending `Foo` it holds that

$$\mathit{object}(\mathit{instanceof}\ \texttt{Bar}) \sqsubseteq \mathit{object}(\mathit{instanceof}\ \texttt{Foo}).$$

The *array* type is extended to specify the possible key and value types. Array key types in PHP can only be *int* and *string*.[2] Value types include all the basic types (without further subdivision for objects or nested arrays), as well as *ref*, which is used if the array may contain references. As an example, for an array that has string keys and numeric values, we write $\mathit{array}[\mathit{string} \to \mathit{int}|\mathit{double}]$.

The ordering for array types can be obtained by considering each variant of "array can have key of type t" and "array can have value of type t" as separate elements in the type universe T, so that the order is again given by the subset relation. We refer to these elements as *array specialization types*. It may further be noted that a naked *array* type, without additional key/value types, implies that the array is empty. Lastly, if there is at least one value type, there must also be at least one key type, and vice versa.

In addition to these proper types, we also track a number of other type-like properties using the same mechanism. Firstly, the already mentioned *ref* type denotes that a variable may be a reference. As we do not perform alias analysis to narrow the possible changes of a reference variable, we require that all lattice values including *ref* also include a union of all other proper types (denoted *any*). As such *ref* may only occur as *any*|*ref* or a superset.

Secondly, the *undef* type denotes that a variable may be uninitialized. An *undef* variable will generally behave like a *null* variable, however the distinction is important because *undef* variables will, under most circumstances, throw a run-time warning when accessed.

The type lattice we use has two main limitations in what it can represent: The first is that array types are only tracked to one level of nesting, i.e. we can only determine that a variable is an "array of arrays", but not that it is an "array of arrays of integers" or similar. The second is that objects can only reference a single class, so it's not possible to track union types like $\mathit{object}(\texttt{Foo}|\texttt{Bar})$ or intersection types like $\mathit{object}(\mathit{instanceof}\ \texttt{Foo\&Bar})$. However, both limitations are caused by concerns over the efficiency of the in-memory representation of types, rather than being fundamental limitations of the approach.

The $\mathit{Value}(\cdot)$ relation of the propagation framework is initialized to $\emptyset$, i.e. the bottom element of the lattice, for all variables apart from the implicitly defined variables at the start of the entry block. These are instead initialized to *undef* in functions and *undef*|*any*|*ref* for pseudo-main code, to account for the fact that arbitrary variable values may be inherited from the surrounding scope.

4.2.2. Join operator and transfer function

With the lattice defined, we can consider the remaining components of the propagation framework, in particular the transfer function $\mathrm{eval}(I, v)$. However, before doing so, we

[2] Arrays in PHP are ordered dictionaries implemented using hashtables. It is possible to mix integer and string keys in the same array.

wish to make the behavior of the join operator, which was already implicitly defined through the lattice order in the previous section, more explicit here.

For non-object types the join operation is simply given by a set union $\cup$ (with array specialization types considered to be distinct elements). For object types further consideration is needed: The join between two object types is given by

$$\mathit{object}(A) \sqcup \mathit{object}(B) = \begin{cases} \mathit{object}(A) & \text{if } A = B \\ \mathit{object}(\mathit{instanceof}\ \mathrm{LCA}_u(A, B)) & \text{if } \mathrm{LCA}_u(A, B) \text{ exists} , \\ \mathit{object} & \text{otherwise} \end{cases}$$

where $\mathrm{LCA}_u(A, B)$ refers to the lowest common *unique* ancestor of A and B in the inheritance graph. The emphasis on uniqueness is important here, because, while PHP does not support multiple inheritance, it does support the implementation of multiple interfaces. As such multiple interfaces may be lowest common ancestors (LCAs) of two classes. In such cases it would *not* be legal to choose an arbitrary LCA, as this would make the join operation non-associative. Instead the computation of the LCA needs to be iterated (by computing the LCAs of the LCAs etc.) until either there is a unique LCA or no common ancestor exists.[3] The join of *object* types where at least one operand has an *instanceof* qualifier works the same way, with the difference that the result type will also have the *instanceof* qualifier.

The transfer function $\mathrm{eval}(I, v)$ returns the new lattice value for variable v defined by instruction I based on the current state of the variable-to-lattice mapping $\mathit{Value}(\cdot)$. This function needs to be defined for all instructions that may define variables, although a pessimistic default implementation of returning *any|ref* may be used for cases where more specific information is not available or worthwhile. In the following, we will discuss the implementation of $\mathrm{eval}(I, v)$ for one specific example, namely the `ADD` instruction.

First, we define the *effective type* of an operand as the type with *undef* replaced by *null*, and any specialized information about arrays or objects removed. Given this, the result type of an `ADD` instruction may be determined as follows:

1. Let t_1 be the effective type of the first operand, and t_2 the effective type of the second operand. Further initialize the result type t_r to be empty.
2. If t_1 and t_2 both contain *array*, add *array* and the union of the array specialization types of both operands to t_r.
3. Remove the *array* type from both t_1 and t_2. If this leaves either t_1 or t_2 empty, return.
4. If t_1 or t_2 contain *double*, add *double* to t_r.
5. Remove the *double* type from both t_1 and t_2. If this leaves either t_1 or t_2 empty, return.

[3] In practice we forgo this relatively complex iterated LCA search, and instead compute the LCA over the class hierarchy only. Support for union object types would resolve this issue as well.

6. If t_1 and t_2 are subtypes of $null|bool|int$ and the value range of the result variable does not admit over- or underflow, then add int to t_r.
7. Otherwise add $int|double$ to t_r.

Construction of the transfer function such that it is both always monotonic and generates the smallest possible set of result types turns out to be less trivial than it may initially appear, as such we will discuss some of the subtleties involved in this particular example:

Firstly, it should be noted that the result type only contains *array* if *both* operands may be *array*. A particularly interesting case occurs if the type of one operand is *exactly array*, while the other does not contain *array*. In this case the final result type will be empty.

This is a valid result and indicates that this instruction will *always* throw, thus never returning a proper value that might have a type. All instructions in the basic block following it will be unreachable, and if all transfer functions are minimal, the empty type will be propagated through using instructions. Empty result types could technically be used to eliminate additional unreachable code, but applicable cases are unlikely to occur in practice, so we did not further pursue this.

Secondly, the result type contains *double* if *either* of the operands may be *double*. The reason is that addition with a double will always promote the other operand to double and consequently yield a double result. If one the operands is *exactly double* (checked in step 5), then the result type can only be *double*, otherwise we need to consider further.

Lastly, the types *null*, *bool* and *int* behave like integers under addition. If value range inference determined that the result cannot overflow, the result type will be *int*. Otherwise, if either an overflow to double is possible, or a different type like a string (which may behave as either an integer or double) is involved, then both *int* and *double* are possible results.

4.2.3. Flow-sensitivity: Feasible successors

Because branches on the *null* type, as well as the *true* and *false* pseudo-types, can be determined statically, it is possible to make use of the feasible successor mechanism of the propagation framework to lend a degree of flow-sensitivity to the type inference algorithm.

For an ordinary branch with targets b_t (true) and b_f (false) acting on a variable with effective type t, the feasible successors are given by:

$$\text{feasible-successors}(\cdot) = \begin{cases} \emptyset & \text{if } t = \emptyset \\ \{b_t\} & \text{if } t = true \\ \{b_f\} & \text{if } t \sqsubseteq null|false \\ \{b_t, b_f\} & \text{otherwise} \end{cases}$$

Similar static decisions are also possible for some other branch types, e.g. for null-coalesce branches it may be possible to determine the branch target based on whether the effective type contains *null*.

Under which circumstances can we expect to successfully determine a branch target with type information only? An obvious case are boolean constants occurring directly in the code or being introduced by function inlining. However, even if precise values are not known, the outcome of certain operations may be determined using type information, for example:

1. Checks like `is_int(`v`)` may only be true if *Value*(v) contains *int* and only be false if *Value*(v) contains types other than *int*. (Both may apply, in which case we have no knowledge.)
2. Type-strict comparisons v_1 `===` v_2 may only be true if the types of both operands have a non-empty intersection.
3. Instanceof operations v `instanceof Foo` for a known `Foo` may only be true if *Value*(v) $\sqsupseteq$ *object*(`Foo`).
4. The result of a logical and v_1 `&&` v_2 is false if either operand can only be null or false.

At least in theory, even if a particular case could also be handled by constant propagation, it may still be advantageous to already handle it during inference, as types added by the code segments thus marked unreachable do not have to be considered. The interaction of type inference and constant propagation will be more closely discussed in section 4.3.4.

4.2.4. Flow-sensitivity: Pi type constraints

Next to the feasible successors mechanism of the propagation framework, another means by which we may improve flow-sensitivity is given by the use of π-nodes, which were first introduced in section 3.6.5. We will now discuss in more detail how π-nodes integrate with type inference.

As a reminder, π-nodes are inserted along control-flow edges that are guarded by conditionals checking some type-related property, such as `is_int($var)` or `$var === null`. The purpose of the π-node is to split the SSA variable and associate the property implied by the conditional with it.

For a node $v_1 = \pi(v_0)$ with a type constraint t, the result type of v_1 is given by constr(*Value*(v_0), t), where the constr($\cdot$) function is similar to, but not identical with, the meet operation $\sqcap$ induced by the $\sqsubseteq$ relation as defined previously. For simple types the constraint function is indeed given by a set intersection, however specialized object types require additional handling. We define:

$$\text{constr}(\textit{object}(\textit{instanceof } A), \textit{object}(\textit{instanceof } B)) =$$
$$= \begin{cases} \textit{object}(\textit{instanceof } A) & \text{if } A \textit{ instanceof } B \\ \textit{object}(\textit{instanceof } B) & \text{if } B \textit{ instanceof } A \\ \emptyset & \text{if } A \text{ and } B \text{ are classes} \\ \textit{object}(\textit{instanceof } A) & \text{otherwise} \end{cases}$$

The first two cases are clear: If one class is a subclass of the other, we use the more specific one (this includes the case where both are equal). However if A and B are not in an inheritance relation, we encounter a problem: As PHP does not support multiple inheritance, this condition is not satisfiable if A and B are classes, in which case the result is empty. However PHP does support implementing multiple interfaces, so that two unrelated interfaces A and B may still both be implemented by one class.

There are two issues with this: First, there may be more than one class implementing the two interfaces. Second, not all classes are known at compile-time, i.e. there may be no knowledge of a class implementing both interfaces, even though it will exist at runtime. In either case, the meet operation $\sqcap$ induced by the $\sqsubseteq$ relation would return $\emptyset$, which does not constitute a correct analysis result.

We resolve this issue by simply choosing the left-hand type $\mathit{object}(\mathit{instanceof}\ A)$ as the result type. The left-hand type is used on the principle that SSA form with π-nodes should never yield a worse result than SSA without π-nodes. For the case where the left-hand *object* type specifies an *exact* type, the constr$(\cdot)$ function simplifies to:

$$\text{constr}(\mathit{object}(A), \mathit{object}(\mathit{instanceof}\ B)) = \begin{cases} \mathit{object}(A) & \text{if } A \ \mathit{instanceof}\ B \\ \emptyset & \text{otherwise} \end{cases}$$

4.2.5. Type narrowing

As already discussed in the introduction, our approach to type inference is purely forward directed: The type of a variable will never be affected by the operations it is used in. We have justified this using the circumstance that we do not have the option of constraining the type of parameters or other sources. However, it *may* be possible to change the type of a constant variable initialization, as this is not an externally provided value. In the following we will describe a mechanism we refer to as *type narrowing* which does precisely this for a specific case.

To motivate this additional mechanism, consider the control flow graph in Figure 4.1, which is a small excerpt of a Mandelbrot set computation. The code first initializes the variables z_1 and Z_1 to integer zero. However, the new variables z_3 and Z_3 computed by the while loop will always be doubles (assuming the variables i_1 and r_1 are doubles). As a result, the variables z_2 and Z_2 will have a union type of $\mathit{int}|\mathit{double}$.

This union type is problematic, as it precludes the use of type specialized instructions down the line. For a JIT compiler the situation is even worse, as the union type will prevent unboxing of the value. This problem only exists because the programmer did not initialize the variables with the correct type (using an integer zero instead of a double zero). Our goal is to detect such cases and promote the initialization to use doubles, if doing so causes no change to observable results.

Promoting an initializer to double means that computations using this value will use double arithmetic instead of integer arithmetic, which may lead to a loss in precision. As such, changing the type of the initializer is only possible if we can prove that earlier use of doubles has no impact on results.

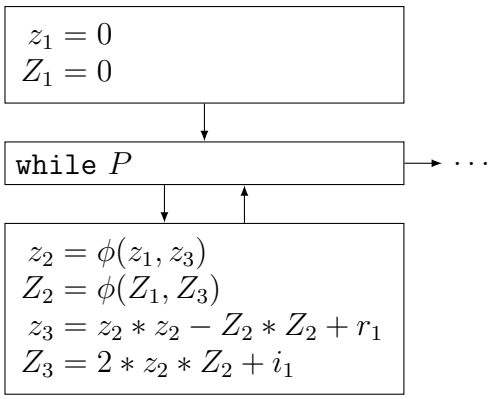


Figure 4.1.: Excerpt from a Mandelbrot function. The variables z_1 and Z_1 are initialized to integer zero, while the variables z_3 and Z_3 are known to be double (if r_1 and i_1 are doubles). As a result z_2 and Z_2 have a union type *int|double*. By promoting the z_1 and Z_1 initialization to double, this can be avoided.

Type narrowing is performed as follows: For each assignment of an integer literal to a non-reference variable, we use the canPromote function shown in Listing 4 to determine if the assignment may be promoted to double. As a side-effect, this function also computes the set of variables those type information may change due to the promotion (*visited*). The type information for these variables is reset to be empty and the assignment instruction is added back to the type inference instruction worklist. Once all instructions have been processed we rerun type inference (if at least one promotion occurred).

The canPromote function recursively[4] checks that a double promotion will not change the result of instructions using the variable. This is essentially done by passing the integer value of the currently considered variable along and checking whether computations still yield the same result if it is replaced with a double value. For a use of the variable in a non-ϕ instruction the function proceeds roughly as follows:

1. Ignore the use if it improper, i.e. if the use occurs as the target operand of an assignment, as described in section 3.6.4.
2. If the instruction is not *narrowable*, promotion cannot occur. By narrowable we refer to arithmetic instructions which return a double result if one operand is a double. For example this includes the addition, subtraction, multiplication, division and power-raising operations, but does not include the modulus instruction, which in PHP always produces integers.
3. If the result type of the instruction is exactly *double*, the variable will be cast to double in any case. As such, promotion is allowed (as long as other uses do not disagree).
4. If all the operands of the instruction are known, because they are either the currently considered variable or constants, we can evaluate the instruction using the original

[4] An iterative formulation using a worklist is of course also possible. We present a recursive version here for simplicity.

Listing 4 Main component of type narrowing algorithm

```
function canPromote(variable var, original value value)
    ▷ visited is a set shared between invocations and initialized to ∅
    if var ∈ visited then
        return true
    visited ← visited ∪ {var}

    for all I ∈ uses(var) do
        res ← result variable of I
        if I is φ-node then
            if ¬canPromote(res, value) then
                return false
        else if use of var in I is improper then
            continue
        else if I is not narrowable then
            return false
        else if result type of I is exactly double then
            continue
        else if value = CAST then
            return false
        else if operands of I are known then
            ival ← result of computation on original values
            dval ← result of computation on doubles
            if double(ival) ≠ dval then
                return false
            if ¬canPromote(res, ival) then
                return false
        else if I is effective cast then
            if ¬canPromote(res, CAST) then
                return false
        else
            return false
    return true
```

operands and using doubles, and check whether the result is the same (apart from type).

5. If not all operands are known, it may still be possible to show that the eventual result is the same, if the instruction only acts as an effective double cast. For example `1.0 * $var` is the same as `(double) $var`. If the instruction that consumes this result would cast the operand to double anyway, performing the cast earlier will not change the result.

6. If either of the last two cases apply, we recursively check whether the result variable

can be promoted, passing either the result of the calculation as the new value or using a special CAST placeholder.

Applying this to the example from Figure 4.1, we start with the initialization of z_1 to integer zero. z_1 is used in a ϕ-node producing z_2. We recursively check z_2, which is used in $z_2 * z_2$, with result zero in both integer and double arithmetic. This result is then used in the expression $result - Z_2 * Z_2$. An expression of type $0.0 - v$ acts as an effective cast, because in PHP it holds that `(double)(0-$v)` and `0.0-(double)$v` are always bitwise identical.[5] This result in turn is used in $result + r_1$ with r_1 being a double. As this would cast the *result* operand to double, the previous effective cast is allowed.

Similarly, starting at $Z_1 = 0$, this variable is used in a ϕ-node with result Z_2. This variable in turn is used in an expression of type $var * Z_2$. As Z_2 is zero at this point, the result will always be zero independently of the value of *var*. This result is then used in $result + i_1$, where i_1 is a double. As the results are consistent until this point and i_1 would force a double cast, an earlier promotion is allowed here as well.

The type narrowing procedure uses an ad-hoc approach to solve a specific problem, lacking any particular generality or elegance. The main reason why it is effective is that mis-typed initializers tend to use values like zero or one, which have special properties we can exploit. For our particular example an alternative solution would be to start at a narrowable instruction returning a double and walk the SSA graph backwards to the variable initializations and require that *all* of them be promotable (here z_1 and Z_1). This has the advantage that we would be able to simply perform all computations using the value of both z_1 and Z_1 without resorting to any particular "tricks". On the other hand this would not cover cases where only one variable can be promoted.

4.3. Constant propagation

Constant propagation is an analysis, which aims to detect variables that are constant for all possible executions of a program. This allows the replacement of uses of constant variables with constant instruction operands, and the subsequent elimination of the generating instructions, thus yielding programs that are smaller and faster.

On SSA form, this problem can be solved efficiently using the sparse conditional constant propagation (SCCP) algorithm due to Wegman and Zadeck [33]. We will rephrase this algorithm in terms of the general propagation framework introduced in section 4.1 (which itself is just a generalization of the SCCP algorithm).

In the following subsections, we will first introduce the constant propagation lattice and define an instance of the propagation framework based on it. Then we will discuss a number of PHP specific issues and lastly consider how constant propagation may be combined with type inference.

[5] This fact is non-trivial and depends on the specific behavior of subtraction in PHP. In particular `0-PHP_INT_MIN` is specified to result in `0.0-(double)PHP_INT_MIN`. The equality would not hold in C.

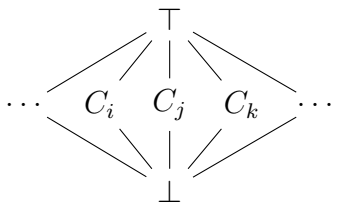


Figure 4.2.: Hasse diagram for the constant propagation lattice.

4.3.1. Constant propagation lattice

The constant propagation lattice $(L, \sqsubseteq)$ shown in Figure 4.2, consists of three kinds of elements: A top element $\top$, a bottom element $\bot$ and an infinite number of constant elements C_i in between. While it holds that $\bot \sqsubseteq C_i \sqsubseteq \top$ for any C_i, the constant elements C_i have no ordering among themselves, i.e. for any $i \neq j$ both $C_i \not\sqsubseteq C_j$ and $C_j \not\sqsubseteq C_i$. The meet operation $\sqcap$ on this lattice is thus given by:

$$\begin{aligned} \top \sqcap l &= l && \forall l \in L \\ \bot \sqcap l &= \bot && \forall l \in L \\ C_i \sqcap C_j &= C_i && \text{if } i = j \\ C_i \sqcap C_j &= \bot && \text{if } i \neq j \end{aligned}$$

When associated with an SSA variable, the different lattice values are to be interpreted as follows: The $\top$ element represents an *underdefined* variable, which may or may not be constant. We do not yet know a specific constant value this variable may have, and neither do we know that it is non-constant. Conversely, the $\bot$ element represents an *overdefined* variable, which cannot be proven to be constant, either because it depends on an external source or because we have observed that it may take multiple distinct constant values. The C_i values of course represent a variable that is constant with value C_i, at least according to the current state of the algorithm.

The idea behind this lattice is that variables will be optimistically initialized to the $\top$ state, may then lower to a constant value C_i and then lower again to $\bot$, if it turns out that this is not the only constant value the variable may take. After the propagation framework has reached a fixed point, all variables (with reachable definitions) will have either a C_i or $\bot$ value. Uses of variables in the former category may then be replaced with the constant value.

4.3.2. Transfer function and feasible successors

The transfer function $\text{eval}(I, v)$ of the propagation framework returns the new lattice value for variable v defined by instruction I based on the current state of the $\mathit{Value}(\cdot)$ relation. For constant propagation this function essentially performs a compile-time evaluation of I, with special semantics for the $\top$ and $\bot$ elements. For simplicity, let us consider an instruction of the form $I : v = v_1 \text{ op } v_2$ and let $l_1 = \mathit{Value}(v_1)$ and $l_2 = \mathit{Value}(v_2)$. Then $\text{eval}(I, v)$ behaves as follows:

1. If either l_1 or l_2 is $\bot$, or if the instruction cannot be statically evaluated in general, then the result is $\bot$.
2. Otherwise, if either l_1 or l_2 is $\top$, the result is $\top$.
3. Otherwise both l_1 and l_2 must be known constants. If the operation can be evaluated for these particular constants, let that be the result.
4. Otherwise the result is $\bot$.

Intuitively this means that if one operand is non-constant, the result is also non-constant. Similarly if we don't know whether one operand is constant, we also don't know whether the result is constant.

For some specific operations it may be possible to determine a constant result even if one of the operands is not known. A typical example is the expression $v * 0$, which always evaluates to 0, independently of the value of v. If one wishes to exploit this, care must be taken when defining the interaction with $\top$ and $\bot$ to ensure the transfer function is still monotonic. If one defines that $l * 0 = 0$ for any $l \in L$, then one should also chose that $\top * \bot = \top$ rather than $\bot$. Otherwise the function becomes non-monotonic if the $\top$ operand transitions to 0.

The last component of the propagation framework we need to define is the edge feasibility function. For a conditional branch on variable v we determine the feasible successors as follows: If $\mathit{Value}(v) = \bot$ all successors are feasible. If $\mathit{Value}(v) = \top$ no successors are feasible. Otherwise $\mathit{Value}(v)$ is a constant and there will be one feasible successor.

4.3.3. Specifics of constant propagation in PHP

The $\mathit{Value}(\cdot)$ relation is initialized to $\top$ for all variables apart from the implicit variables at the start of the entry block, which are initialized to $\bot$ instead. Alternatively it would be possible to initialize the latter to constant `null`, as undefined variables evaluate to the value `null` upon use. However, access to undefined variables additionally throws a run-time warning, so that it would not be possible to actually substitute this value at the use-sites, making this initialization of little use.

It should also be noted that variables, which type-inference has determined to be potential references, are *not* initialized to $\bot$ as might be expected: If a variable is a reference its value might change unexpectedly, e.g. through a modification in an error handler. As such propagating a constant assigned to a reference is generally not safe. However, initializing potential reference variables to $\bot$ at the start of the analysis is overly pessimistic, as is illustrated in Figure 4.3. In this example type inference will determine b_2 and b_3 to be potential references. However constant propagation shows that the block containing the b_2 reference assignment is not reachable, so that $b_3 = b_1 = 1$ would be a valid analysis result.[6]

Instead we initialize potential references to $\top$, like all other variables, and rely on the fact that any instruction producing a reference will yield a $\bot$ value for the corresponding

[6] As we have to assume that any call to an unknown function turns its arguments into references, this situation is more common than one might expect.

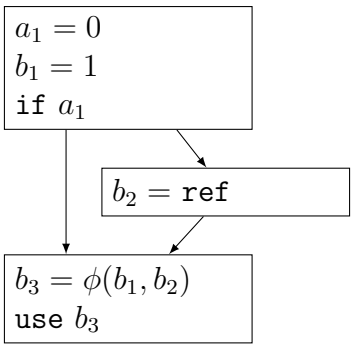


Figure 4.3.: Example CFG where initialization of potential reference variables to $\bot$ produces suboptimal results: The branch containing the reference assignment to b_2 is shown to be unreachable during constant propagation.

variable. However this requires additional handling for ordinary assignments of the form $(v_1 \to v_2) = w$ (where v_1 denotes the improper use of the old value and v_2 the new value): Normally a value of $\bot$ for v_1 may be ignored, as it has no bearing on the new value of v_2. However, if v_1 is marked as a potential reference, we cannot distinguish whether $\bot$ refers to a non-constant value or a reference, so we pessimistically assume it is a reference and set v_2 to $\bot$ as well.[7]

With the issue of references covered, we will now discuss a few PHP specific issues relating to the implementation of the transfer function. First, it should be noted that many basic operations in PHP will generate either a run-time warning or exception when used with certain operand types or values. For a semantically identical transformation these errors need to be preserved. For the case of run-time warnings, it would be possible to still compute the result of the operation and use it as the input to dependent operations. However, this would complicate further handling, as the implementation would have to remember that this value was generated along with a run-time warning and as such its generating instruction may not be eliminated. As it is unlikely that code generates run-time warnings in the course of ordinary execution, we did not consider this to be worthwhile and instead return a $\bot$ value for erroring operations.

The second potential issue to consider are the types of values that may participate in constant propagation in PHP. In a classical setting, constant propagation is performed on booleans, integers and doubles, or similar simple scalar types. However, in PHP constant propagation may also be performed on more complex types like strings and arrays. This poses a problem for the complexity of the algorithm: While the transfer function can still only be evaluated a linear (in the size of the SSA graph) number of times, we can no longer assume evaluations of the transfer function to be cheap.

A particular striking example of this issue is the instruction sequence generated for the construction of an array. It consists of an `INIT_ARRAY` instruction followed by one `ADD_ARRAY_ELEMENT` instruction for each element of the array. In non-SSA form these

[7] Alternatively, it is also possible to extend the lattice by introducing a $\bot_r \sqsubseteq \bot$ value specifically indicating a reference. As discussed in section 4.3.4, combining type inference and constant propagation also solves this.

instructions operate on a shared result variable that is incrementally modified. In SSA form however, each of the instructions in this chain will produce a new SSA variable. When applying the constant propagation algorithm to this case (assuming all elements are known) a naive implementation will assign separate arrays with an increasing number of elements to each variable. For the construction of an array with N elements, this degrades complexity to $O(N^2)$ both in terms of execution time and memory usage.

This particular case can be resolved by exploiting the structure of the chain: Each of the intermediate SSA variables is used only in the subsequent `ADD_ARRAY_ELEMENT` instruction, which does not admit constant substitution. As such, we can safely share one array among all instructions (or use some other dummy value in all but the most recent instruction), as long as we avoid reevaluations with unchanged operands. However, this solution is not generally applicable. For example, if the array is constructed through manual insertions rather than the use of array literal syntax, the guarantee of a single use does not exist and thus it is not possible to share the value. In this case we may only limit the maximum size of the array, and switch to a $\bot$ result if it is exceeded.

When describing the constant propagation algorithm, it was mentioned that for some operations like $v * 0$ it is possible to determine the result of the operation even if the value of v is not known (i.e. $\top$ or $\bot$). While in principle true, such relations are typically only very narrowly applicable in PHP. For example, the result of `$var * 0` will be double zero instead of integer zero if `$var` is a double. Similarly, the result of `$var * 0.0` is not always 0.0, but may also be -0.0 or NaN for some values of `$var`. Furthermore, if `$var` is a string the operation might generate a warning or, if it is an array, might even generate an exception. As such, exploiting multiplication by zero and similar relations is only safe if type information is taken into account.

After the constant propagation algorithm has finished executing, we make use of the results by replacing uses of constant variables with their value. A problem encountered here is that many PHP instructions have restrictions on the use of constant operands. Some instructions do not support constant operands outright, while others require specific types or adjusted values. For this reason, even if the value of variable is known, we may not always make use of it.

After constant operands have been replaced, we perform a simple dead code elimination (DCE) pass, which walks instructions in postorder and removes them if they generate temporaries which are both unused and have a known value. If they generate non-temporary variables instead, the instruction is replaced by a constant assignment, thus deferring actual elimination to the main DCE pass.

The motivation for implementing a primitive DCE pass used only for constant propagation is twofold: Firstly, the main DCE pass uses type information to determine whether an instruction may generate an error and thus cannot be eliminated. As such, it would not be able to eliminate instructions that might error for the inferred types, but do not generate an error for the specific propagated constant operands. Secondly, for temporaries the using instruction is also responsible for destroying the temporary. If the use of the temporary is replaced by a constant but the generating instruction not removed, this may result in a memory leak. As we wish constant propagation to be usable independently of the main DCE pass, this primitive DCE ensures that no leaks occur.

4.3.4. Combining type inference and constant propagation

It is a common observation that combining two separate analysis passes may yield better results than sequentially running the passes an arbitrary number of times, e.g. the sparse conditional constant propagation algorithm detects more unreachable code and constant variables than running unreachable code elimination and non-conditional constant propagation in sequence (any number of times).

The common propagation framework makes it very easy to combine analysis passes based on it: For two bounded lattices $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ and transfer functions $\text{eval}_1(\cdot)$ and $\text{eval}_2(\cdot)$, as well as feasibility functions $\text{feasible-successors}_1(\cdot)$ and $\text{feasible-successors}_2(\cdot)$, we define a new bounded lattice $(L, \sqsubseteq)$ with $L = L_1 \times L_2$ and

$$(l_1, l_2) \sqsubseteq (l_1', l_2') \quad :\Leftrightarrow \quad l_1 \sqsubseteq_1 l_1' \wedge l_2 \sqsubseteq_2 l_2' \qquad \forall l_1 \in L_1, l_2 \in L_2,$$

as well as

$$\begin{aligned} \text{eval}(I, v) &:= (\text{eval}_1(I, v), \text{eval}_2(I, v)) \\ \text{feasible-successors}(b) &:= \text{feasible-successors}_1(b) \cap \text{feasible-successors}_2(b), \end{aligned}$$

in order to construct a new instance of the propagation framework that subsumes both passes. Both sub-instances will run in parallel, but mostly independently of each other. The only interaction occurs in the feasible-successors$(\cdot)$ function, which now only considers successors that are feasible under *both* instances. For this reason running both instances in parallel in this way is never worse than running them sequentially, but may be better, as additional control flow paths are excluded.

For the parallel execution of type inference and constant propagation in particular there are two ways in which a better result may be achieved: The first is the aforementioned possibility of reducing the number of feasible successors. In the majority of cases the feasible successors as determined by type inference are a superset of those detected by constant propagation, as type inference essentially performs a very limited form of constant propagation on null and booleans only. However, as described in section 4.2.3, there is a small number of cases where we can determine the outcome of branches based on type-inference information, but not using constant propagation.

The second is given by the fact that in our formulation type inference and constant propagation are not fully independent. Namely constant propagation uses information on potential reference variables as determined by type inference when evaluating assignment instructions. As discussed in the previous section, initializing potential reference variables to $\bot$ is overly pessimistic, because all reference-producing paths may be found to be unreachable during constant propagation. However a pessimistic assumption still had to be made for assignments. Running both passes in parallel resolves this problem, because type inference will not take the code-paths determined to be unreachable by constant propagation into account either.

In practice we have found that these benefits do not materialize. While it is possible to construct codes that derive benefit from this combination, we did not find this to occur in practice. However, running both passes in parallel does avoid the need to rerun type inference after constant propagation to receive the most accurate results.

Listing 5 Dead code elimination algorithm

```
InstrWorklist ← ∅
Live ← ∅
for all instructions I do
    if I has side effects then
        Live ← Live ∪ {I}
        InstrWorklist ← InstrWorklist ∪ {I}

while InstrWorklist ≠ ∅ do
    I ← any element of InstrWorklist
    InstrWorklist ← InstrWorklist \ {I}
    for all instructions I' that define operands of I do
        if I' ∉ Live then
            Live ← Live ∪ {I'}
            InstrWorklist ← InstrWorklist ∪ {I'}
```

4.4. Dead code elimination

Dead code is commonly left behind as the result of other optimizations, such as constant propagation. There are two broad classes of dead code: The first is unreachable code, which refers to instructions that can never be reached by control flow. Unreachable code is already detected and eliminated by the conditional propagation passes (such as conditional constant propagation) discussed in the previous sections. The second type of dead code, with which we are concerned here, refers to instructions which have no side-effects and those results are not used, or only used by other dead instructions. In the following we will first introduce the general DCE algorithm and then discuss some PHP specific concerns.

4.4.1. Algorithm

Dead code elimination on SSA form can be performed using a simple worklist-driven algorithm, as shown in Listing 5. It proceeds from an optimistic assumption that only instructions that have side-effects are live and all other instructions are dead. It then propagates the liveness information backwards: If an instruction is live, then any instruction that generates one of its operands must also be live.

In this basic formulation it is required to consider all conditional branches to be live, i.e. have a side-effect. This may yield suboptimal results, in that all code guarded by the conditional branch may be found to be dead by this algorithm, while the now useless branch will still be considered live. There exists an extension of this algorithm due to Cytron et al. [17], which improves handling of conditional branches by making use of control dependence properties:

Without formal definition, a CFG node m is said to be control dependent on a node n, if there is an edge from n that definitely causes m to be executed, and there also exists

a path from n that does not contain m. Cytron et al. have shown that this condition is equivalent to n being part of the *reverse dominance frontier* (RDF) of m, where the reserve dominance frontier refers to the dominance frontier on the reversed control flow graph (dominance on the reversed CFG is called *postdominance* on the ordinary CFG).

Cytron's dead code elimination algorithm extends the naive variant by initially assuming that conditional branches are dead, and adding an additional step when handling newly live instructions: Conditional branches terminating blocks in the reverse dominance frontier of the block containing the newly live instruction must also be marked live.

In our implementation, we chose not to use the extension by Cytron, as we estimate the number of cases where this is applicable to be relatively low, while the computation of a postdominator tree and the reverse dominance frontiers is both expensive and implementationally non-trivial. The reason for the latter is that the reverse CFG, at least in our representation, has some inconvenient properties: Not only may it have multiple exit nodes, it may also, for the case of infinite loops, have none at all.[8]

4.4.2. PHP specific considerations

As usual, there are number of PHP specific considerations that need to be taken into account when applying the DCE algorithm. Firstly, we should define more precisely what we mean by instructions having side-effects. As already discussed, we consider branches (both conditional and unconditional) to have a side-effect (of branching). It should also be clear that instructions such as `ECHO` which perform I/O operations, as well as function calls (at least without more specific knowledge) are considered to have a side-effect.

However, in addition to these, we also need to consider instructions that may potentially throw a run-time warning or an exception as having side-effects. This is problematic because in PHP nearly every instruction can generate an error, if only in an obscure circumstance. Out of approximately two hundred VM instructions, less than ten have no error conditions at all. To reduce the number of liveness roots, we use type information to determine whether the instruction may error for a particular combination of operand types.

For this classification there is one particular error condition that (thankfully) does not have to be taken into account: PHP's instruction set defines that an instruction using a VM temporary is also responsible for releasing it. As this may directly or indirectly trigger the destruction of an object with a destructor, this operation may error. However, if we eliminate the instruction, we will still need to destroy its operands, so this side-effect is preserved and we do not have to consider it here.

Another case that requires special consideration are assignment operations of the form $(v_1 \to v_2) = w$. If v_1 is a potential reference, the assignment may have a side-effect. However there are two further cases where eliminating this instruction may not preserve exact semantics: If v_1 may have a destructor, then removing this instruction might cause the destructor to run later. Similarly, if w may have a destructor, eliminating the instruction

[8] In this case it is still possible to compute a complete postdominator tree, by considering the backedge targets of infinite loops as additional exit nodes.

may cause the destructor to run earlier.

Assignment operations also pose an additional problem: The use of v_1 in $(v_1 \to v_2) = w$ is improper, as such we do not consider it as a use for the purposes of DCE. This means that if we mark the assignment as live we will *not* mark the instruction generating v_1 as live as well. While this approach is acceptable if v_1 is generated by an ordinary instruction, it also implies that ϕ-nodes those result is only used improperly will be considered dead on termination of the algorithm. This is not acceptable, as the improper uses are still uses and removing the 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 we mark all ϕ-instructions that are used improperly as live and propagate this information backwards to the ϕ-sources using the same worklist-based approach. Only after this step will dead ϕ-instructions be removed.

Dead code elimination, as well as other SSA transformations, often leave behind a large number of ϕ-nodes that are not formally dead, but still unnecessary. These *trivial* ϕ-nodes take the form $v_0 = \phi(v_1, v_1)$, or $v_0 = \phi(v_0, v_1)$, i.e. all source variables of the ϕ-node are the same, or are equal to the result variable. Such ϕ-nodes may be eliminated by renaming uses of the variable v_0 to the common source v_1.

4.5. Type specialization

Many instructions of the PHP virtual machine need to implement different behavior depending on the type of the operands. This is usually done using a fast-path/slow-path split, where the fast-path handles the one or two most likely types for the operation, and falls back to the slow-path which, next to a type generic implementation of the operation, also handles unlikely conditions like undefined or referenced variables.

While this split somewhat improves the situation, even the most basic operations still require multiple type checks. For example an `ADD` on two integers requires two type checks, while an `ADD` on two doubles needs three type checks (one more because integers are checked first). Additionally, the existence of a slow-path with function calls leads to the emission of stack management instructions in the prologue and epilogue of the opcode handler.

To avoid this overhead, we can specialize generic instructions to type-specific ones based on the type information determined by type inference. For the `ADD` instruction for example, one may introduce `ADD_INT` and `ADD_DOUBLE` operations which only accept integer/double operands. Additionally taking into account value range inference on the result, one can further specialize `ADD_INT` to `ADD_INT_NO_OVERFLOW`, which removes not only the type checks, but also the addition overflow branch.

Next to the specialization of arithmetic instructions, type information can also be used to elide certain instructions entirely. For example a `CAST` instruction may be removed if the type of the input operand already matches the type that it is being cast to. Similarly `VERIFY_RETURN_TYPE` may be elided if we have inferred that the return type is always correct.

Instead of specializing for specific types, we may also exploit certain common properties. In particular the types *null*, *bool*, *int* and *double* do not use reference counting. This means that an assignment to a variable that may only hold one of these types does not need to destroy the old value, it can be directly overwritten instead. This allows for a number of related optimizations we collectively refer to as *assignment contraction*:

First, we may convert `ASSIGN $var, val` instructions into `$var = QM_ASSIGN val` instructions. The difference between these two operations is that the former will destroy the old value of `$var`, as well as handle other conditions like assignments to references, while the latter will simply overwrite the value of `$var` directly.[9]

More importantly, we may note that code like `$a = $b + $c` will be compiled into a sequence of two instructions: First `T = ADD $b, $c` will write the result of the addition in a temporary `T`, and then `ASSIGN $a, T` will copy this result into the `$a` variable. The operation is split into these two parts to avoid repeating the relatively involved logic of non-temporary assignments in every instruction. If we know that before this operation `$a` holds a value which does not use reference counting, these two instructions may be contracted into `$a = ADD $b, $c`.

When performing this transformation, there is one caveat regarding exception safety that needs to be taken into account: The VM specifies that a throwing instruction is responsible for making sure its own result variable is released. Usually this is not a problem, as instructions will not write a result in the first place if an exception is thrown. In some cases however, most notably for call instructions, it is possible that the result variable is first written and later destroyed due to a thrown exception. Applying the transformation for such instructions might thus lead to a double free.

The last assignment related optimization we can apply is to convert instructions of type `ASSIGN_ADD $a, $b` into `$a = ADD $a, $b` if `$a` is not reference counted. Once again this avoids a number of checks related to in-place modification of variables. Additionally this transformation enables the arithmetic type specialization that was discussed previously to apply.

Finally, when performing operations on objects for which a class type has been inferred, it may be possible to specialize instructions based on it: Object properties are stored at specific offsets from the start of the object. Usually, accessing a property requires looking up the property offset in a runtime cache. However, if we can statically determine the object class type, this offset can be baked into the instruction as an immediate. This does not require knowledge of the exact object type (*instanceof* constraints are allowed), as the offset remain valid for subclasses.

4.6. SSA liveness checks

For the implementation of copy propagation on conventional SSA form, we will need to determine whether an SSA variable is live at a certain program point. Unlike many other

[9] The `QM_ASSIGN` instruction is designed for use with VM temporaries which can always be assumed to be unused when written to. The name originally derives from its use in the implementation of the ternary (QuestionMark) operator.

data-flow problems, liveness analysis does not particularly lend itself to SSA form, due to its backwards directed nature. However, it is still possible to exploit certain structural properties of SSA form and the CFG when determining liveness of variables. For this purpose, we use the fast liveness checking algorithm by Boissinot et al. [8], an overview of which will be provided in the following.

It should be noted that unlike classical iterative liveness analysis, which provides sets of variables that are live-in/out at a certain block, this algorithm only provides an oracle which can answer queries of the form "Is variable v live-in/out at program point p?", but does not compute actual sets of live variables. The algorithm works by precomputing certain information about the structure of the control flow graph, which can then be used to answer such liveness queries efficiently. Notably this implies that the algorithm is robust against changes of the SSA graph: As long as the control flow graph does not change, the precomputed information remains valid.

In the following, we refer to the unique block in which the SSA variable v is defined as D_v and the blocks where v is used as uses(v). In this context a "using" block does not simply refer to a block b which contains an instruction that uses the variable v as an operand. This is only the case for uses in ordinary instructions, but does not apply to ϕ-nodes: For a ϕ-node in block b with v as its i-th operand, the use is semantically *not* located in b itself, but rather occurs along the control flow edge (p_i, b), where p_i refers to the predecessor associated with the i-th operand. As the block/edge distinction is not relevant here, we locate the use in the block p_i instead.

An SSA variable v may now be defined to be *live-in* at a certain block b, iff there exists a path from b to a use of v, that does not contain the definition D_v. As has already been noted earlier, under strict SSA form the definition D_v dominates any use of v. Using this property, we may equivalently say that v is live-in at b iff D_v strictly dominates all blocks on a path from b to a use of v. If instead the path would ever leave the D_v dominated subgraph, it could only reenter it through D_v, i.e. D_v would be contained in the path.

Boissinot's algorithm now concerns itself with how it can be efficiently determined whether a strictly D_v dominated path from b to a use u exists. Assuming b is strictly dominated by D_v there are essentially two cases to consider: u may be *reduced reachable* from b, which means that it is reachable without following back edges. In this case the whole path is certainly strictly D_v dominated. Alternatively u may be reduced reachable from a back edge target t strictly dominated by D_v, which in turn is reachable from b. In this case it has to be ensured that the path from b to t does not leave the D_v dominated subgraph, which is possible through a careful choice of the back edge targets t that are considered.

The algorithm uses two precomputed structures: R_b is the set of blocks that are reduced reachable from b, i.e. all u for which exists a path from b to u that does not contain back edges. The set T_b contains b itself, as well as certain back edge targets reachable from b: The goal is that $T_b \cap \text{sdom}(D_v)$ should only contain those back edge targets that are reachable from b without leaving the D_v dominated subgraph.

To achieve this, the auxiliary set $T_b^{\uparrow}$ is defined as the set of back edge targets t whose sources are reduced reachable from b, but which are not themselves reduced reachable from b. The latter condition prevents that $T_b^{\uparrow}$ contains back edge targets in the D_v dominated

subgraph if b itself is not dominated by D_v. T_b is now simply the transitive closure over $T_b^\uparrow$ starting at b:

$$T_b := \{b\} \cup T_b^\uparrow \cup \bigcup_{t \in T_b^\uparrow} T_t^\uparrow \cup \dots$$

Given the two sets R_b and T_b a variable v is live-in at block b iff there exists a $t \in T_b \cap \text{sdom}(D_v)$ for which $R_b \cap \text{uses}(v) \neq \emptyset$, i.e. for which a use of v is reduced reachable from t.

The set R_b can be computed by walking the CFG in postorder and computing $R_b = \{b\} \cup \bigcup_{s \in \text{succs}(b)} R_s$ at each step. T_b is computed in four steps, starting with empty T_b:

1. Walking the CFG in preorder, set $T_b = T_b \cup \{b\} \cup \bigcup_{t \in T_b^\uparrow} T_t$ if b is a back edge target.
2. For each back edge (s, t) add T_t to T_s.
3. Walking the CFG in postorder, set $T_b = T_b \cup \bigcup_{s \in \text{succs}(b)} T_s$.
4. Add b to each T_b.

Boissinot et al. describe the liveness checking algorithm at the block level only. However, it is easy to extend it to provide liveness information with instruction-level granularity. Listing 6 shows the algorithm we use to determine if a variable v is live-in at a non-ϕ instruction I. In the implementation block(I) refers to the block an instruction occurs in, while $I_1 \geq I_2$ refers to the order of the instructions, which is assumed to be well-defined within one basic block. As presented here, the algorithm also makes the special handling of uses in ϕ-nodes explicit.

To support instruction-level liveness checks two main changes are needed: First, we need to consider the case where the variable v is defined in the same block b as the instruction I at which the check is performed. If the variable is defined after (or at) I, it is certainly not live-in. Otherwise we check if there are uses that are either in a different block or after (or at) I in the same block. For ϕ-uses this is always the case.

Second, for the case where I is in a different block than the definition of v, the ordinary algorithm is used, while only considering uses that are either in a different block than I or after (or at) I in the same block. However, if the block of I is a back edge target we need to consider all uses, as the back edge makes earlier instructions in the same basic block reachable (the source must be reduced reachable if the target is).

The algorithm for live-out checks is the same with $I_1 \geq I_2$ comparisons replaced by $I_1 > I_2$.

For use in PHP two additional adjustments should be made: First, we may ignore improper uses of variables, as well as uses in ϕ-nodes that are only used improperly. Second, if a π-node defines a variable that is immediately consumed by a ϕ-node in the same block, that variable is never live (or rather, it is only live along the control flow edge associated with the π-node, which is a distinction we are not interested in).

Listing 6 Instruction granularity live-in oracle using Boissinot's algorithm

```
function IsLiveIn(variable v, non-φ instruction I)
    b ← block(I)
    I_d ← instruction defining v
    if b = D_v then
        if I_d ≥ I then
            return false
        for all instructions I_u using variable v do
            if I_u is φ-node then
                return true
            if block(I_u) ≠ b ∨ I_u ≥ I then
                return true
    else
        for all t ∈ T_b ∩ sdom(D_v) do
            for all instructions I_u using variable v do
                if I_u is φ-node then
                    for all predecessors p associated with v operands do
                        if p ∈ R_b then
                            return true
                else if block(I_u) ≠ b ∨ I_u ≥ I ∨ b is back edge target then
                    if block(I_u) ∈ R_b then
                        return true
    return false
```

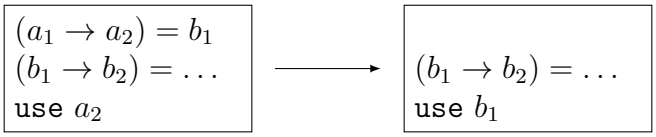


Figure 4.4.: Code before and after ordinary SSA copy propagation. In the new code the *related* variables b_1 and b_2 interfere.

4.7. Copy propagation on conventional SSA form

The goal of copy propagation is to eliminate copy operations of the form $a = b$ by replacing all uses of a with uses of b. On unrestricted SSA form performing copy propagation is very simple, because each variable is defined exactly once, so we do not have to concern ourselves with the possibility of b changing between the assignment $a = b$ and a use of a.

A simple SSA copy propagation algorithm only needs to visit all assignment instructions and can remove them after appropriately renaming uses of the left hand side variable. This process will likely create trivial ϕ-nodes of the form $v_1 = \phi(v_0, \ldots, v_0)$. To remove these in the same pass, we can define copy propagation as an instance of the propagation framework, with the lattice specifying which variable another variable is a copy of, if any.

However, performing copy propagation in this way significantly changes the properties

of the SSA graph. Figure 4.4 shows a sample code before and after copy propagation performed in this manner. For the code before copy propagation, it was possible to translate out of SSA form simply by dropping the variable indexes (and ϕ-nodes in more complex examples). After copy propagation, this is no longer possible as the use of b_1 would then refer to the value of b_2.

This difference can be formalized by introducing a concept of *related* SSA variables, which is 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. For the example in Figure 4.4 the variables a_1, a_2 as well as b_1, b_2 are related. For a ϕ-node $c_3 = \phi(c_1, c_2)$ the variables c_1, c_2, c_3 would be related. This partitions the SSA variables into equivalence classes.

Further, we say that two variables *interfere*, if one is live-out at the definition point of the other. If no variables within one equivalence class interfere, the SSA form is said to be *conventional*. If additionally the equivalence classes coincide with the original non-SSA variables, then translation out of SSA form can be performed simply by dropping all variable indexes and ϕ-nodes. Otherwise the SSA form is said to be *transformed* and the use of an SSA destruction algorithm is required.

Unlike all other optimizations that have been discussed previously, copy propagation can cause conventionality to be lost. For the example in Figure 4.4 the related variables b_1 and b_2 interfere after copy propagation, because b_1 is live-out at the definition point of b_2.

Originally an implementation of the out-of-SSA translation algorithm by Boissinot et al. [7] was planned. However, we encountered many issues when trying to implement it, the most significant of which was that we lose a lot of control over the lifetimes of values stored in variables. Even if we discount concerns about differences in observable destructor behavior, extending the lifetime of a PHP value by introducing an additional copy can have catastrophic effects on performance, because it may trigger subsequent copy-on-write duplications of large structures (in some unfortunate cases causing asymptotic slowdowns). While this and other problems can likely be solved, we decided that further pursuing out-of-SSA translation is not worthwhile at this point, and we will instead require SSA form to stay conventional at all times.

To ensure that copy propagation maintains conventionality, some additional considerations are necessary. For an assignment $(a_1 \to a_2) = b_1$ it should first be noted that while proper uses a_2 should be replaced with b_1, improper uses need to be replaced with a_1 instead, as improper uses refer to the previous value of the assigned-to variable, which will be a_1 after the assignment has been removed.

To guarantee that after copy propagation equivalence classes still coincide with non-SSA variables, it needs to be ensured that a_2 is not used in ϕ-nodes (unless their result variable is only used improperly) and that there are no in-place modifications of a_2, such as $(a_2 \to a_3) \mathrel{+}= 1$ (again excluding improper uses).

Lastly, to ensure interference-freedom within the equivalence class of b_1, the variable a_2 should not be live-out at any modification point of b_1 (whereby we mean any use of b_1 that defines a new b_i on the same operand) or live-in at any block that contains a ϕ-node using b_1.

Of course, if a_1 is a potential reference, copy propagation cannot be used. Additionally the same concerns as for the dead code elimination of assignments apply: If a_1 can have a destructor, removing the assignment may delay its execution. If b_1 can have a destructor, removing the assignment may trigger the destructor too early. If destruction semantics should be strictly observed, copy propagation cannot be used in both cases.

4.8. Function inlining

Function inlining refers to the act of replacing a call to a function with an integrated copy of the function. The motivation for this is twofold: Firstly, procedure calls have overhead, in PHP even more so than in statically compiled languages. Secondly, inlining often leads to additional optimization opportunities, e.g. because constant parameters can be propagated.

While not a primary objective of this work, we have implemented a basic function inlining pass, particularly to judge to what degree it impacts further optimizations. Inlining is performed on raw instruction streams, before construction of SSA form. Calls to inlined functions are replaced by their bodies, with parameter receiving instructions replaced by assignments and returns replaced by assignments to a temporary and jumps to the end of the inlined segment.

Additionally an `UNSET_VAR` instruction is emitted after the inlined function body for each variable that is used in the inlined function. This is necessary for two reasons: First, it prevents values (which may be have observable destructors) from living longer than they would in a non-inlined variant. Second, and more importantly, unsetting all variables ensures that references are broken. If the variables weren't unset and the inlined function called in a loop, a reference from a previous iteration might survive to the next iteration, which would result in different behavior.

Another caveat applies when inlining methods: As described in section 2.7, for reasons of backwards compatibility it is possible for `$this` to be undefined inside instance methods. Inlining a method call on `$this` could thus remove the error that would normally occur on undefined `$this` access. For this reason we insert an additional `ENSURE_HAVE_THIS` instruction, whose sole purpose is throwing an error if `$this` does not exist.

The hope is that further optimizations will eliminate the various additional instructions inserted by inlining: Copy and constant propagation are likely to elide the assignments to the parameter variables. Dead code elimination may remove the additional unsets. CFG simplification may remove jumps inserted to handle function returns. Dominator tree propagation (described in section 4.9) can eliminate unnecessary ensure-have-this instructions.

However, inlining also causes a number of additional issues: From a program semantics perspective, inlining is problematic because it modifies debug and exception stack traces. As such it is likely not suitable as a default optimization without further work in this direction. Additionally, there are two potential performance problems: The first is that inlining increases the program size and consequently also increases the number of CPU cache misses. The second issue is that during inlining the variable spaces of multiple

functions are merged. This is not a concern for VM temporaries, as PHP is able to compact the number of temporaries to insignificant sizes. However, we currently do not have a register allocator for real variables. As VM slots for real variables need to be fully initialized on entry into the function and destroyed on exit from it, this incurs a significant overhead if the number of variables is increased (and does so even if the inlined code-path is never executed).

Lastly, there is only a limited number of functions that are eligible for inlining. While some functions cannot be inlined due to use of dynamic features, the more significant issue is posed by the single-file view of the current PHP compiler. As such, only functions in the same file may be inlined, severely limiting the applicability. For object oriented code, an additional complication is that we can typically only be certain about the called method if it is private or final. As we did not implement speculative devirtualization, this further limits the number of inlinable procedures.

As inlining was not a primary goal of this work, we did not investigate different inlining heuristics in detail. For the results in section 5.2 we tried both a conservative heuristic, which only inlines small functions with many constant arguments, as well as an aggressive heuristic, which inlines all eligible functions (that are not excessively large) to one level.

4.9. Propagating information along the dominator tree

In some cases it is possible to propagate useful information, and optimize based on it, using only the dominator tree, without requiring or benefiting from SSA form.

If n dominates m, then n is always executed before m. As such, if n enforces a certain condition (by throwing if it is not met), we know that this condition holds in m as well. The dominator tree can be used to efficiently propagate this information:

We traverse the dominator tree starting from the root. If, while walking the instructions of a block, a condition is enforced, we push this information onto a stack and can make use of it while it is on the stack. Before recursively processing child nodes, we remember the current position on the stack. After a child node has been processed, we discard all information after this position.

Currently, we use this method for two purposes: The first is to reduce the number of function arguments for which we have to assume that by-reference passing is used, because we do not know the function signature (defined in a different file). This is done by exploiting the fact that passing literals and some expression types to a by-reference argument generates an exception. If we see that an argument of an unknown function is used in this way, we can conclude for all dominated code that this argument uses by-value passing. The second is to remove redundant `ENSURE_HAVE_THIS` instructions, if it has already been ensured that `$this` exists.

It should be noted that dominator tree propagation is only a cheap and easy to implement approximation. For example, it is not able to handle diamond control flow, i.e. detect that if a condition is enforced in both branches of an if/else structure the condition will also hold when control flow rejoins. To accurately handle this, one would have to rephrase the problem in terms of data-flow equations.

4.10. Testing and verification

In the course of this chapter a number of different optimizations has been introduced. It is well known that writing optimizing compilers that are also correct is a non-trivial task, so we wish to briefly describe which steps we have taken to gain at least some degree of confidence in our implementation.

The PHP testsuite includes approximately fifteen thousand end-to-end execution tests, an estimated three thousand of which are testing core language functionality. A problem we ran into early on, is that the majority of these tests run in the pseudo-main scope and as such are effectively excluded from optimization. This means that even blatantly wrong optimizations sometimes passed the testsuite.

To counter this problem, we have built an automated test porting tool based on our PHP-Parser project, which moves non-declaration code into a dummy function and calls it. Because this has to exclude tests that use global variables or have line number dependent output etc. we can only port approximately half of the core language tests in this manner. While this is not very extensive, it does, in combination with tests on real PHP projects, provide a baseline.

In addition to this, we verify that all invariants of the used SSA implementation are maintained between passes, and after optimization has finished, also verify that all SSA variables associated with one non-SSA variable are interference-free (ensuring our requirement of conventionality). While this makes no direct statement about the correctness of the transformed IR, it does avoid certain categories of bugs.

Lastly, in order to verify the results of type inference, we have implemented a mode in which a type assertion instruction is emitted after any program point defining a new SSA variable. The testsuite is then run with the bytecode instrumented in this manner to detect discrepancies between type inference results and actual types. We have found this approach to be extremely valuable in finding edge-case bugs in the inference transfer function, as well as the table of internal function return types.

5. Results

In this chapter, the performance impact of the optimizations discussed in the previous chapter will be measured and analyzed. We will first consider the effect on various microbenchmarks shipped with the PHP distribution and break down the impact of individual optimizations. Subsequently, we consider the effect on real applications, using WordPress and MediaWiki as examples, and discuss why some optimizations are effective, while others are not.

All measurements were performed on an Intel Core i5-4690 CPU in a virtualized Ubuntu 14.04 environment. The numbers presented are averages over many executions.

5.1. Microbenchmarks

The PHP implementation ships with two sets of standard microbenchmarks. The first (bench.php) implements a number of functions that either perform simple algorithms (e.g. computations of Mandelbrot sets or Ackermann numbers) or certain code pattern (e.g. writing to and reading from arrays in specific orders).

Figure 5.1 shows the normalized execution times of these microbenchmarks for three scenarios: The baseline refers to use of the previously existing optimization pipeline. The other two results both use our additional data-flow optimizations and selectively enable inlining. The geometric mean speedup is 1.20× without inlining and 1.42× with inlining.

There are a few things we can conclude from these results: First, there are a number of benchmarks which aren't affected by our optimizations at all, including the majority of the array population benchmarks. This is not particularly surprising as we do not perform array related optimizations.

Second, most benchmarks are not affected by inlining, as they only use a single function or few calls. Large effects can be seen for simpleu(d)call, where inlining enhances dead code elimination. The Fibonacci number benchmark also benefits from inlining one level of recursive calls.[1] On the other hand, the computation of Ackermann numbers suffers a minor regression if recursive calls are inlined. This is a symptom of the issue discussed in section 4.8, namely an increase in the number of variables for code-paths that are not executed.

For the benchmarks where our optimizations were effective, Figure 5.2 shows a more detailed breakdown of which optimizations contributed to the reduction in execution

[1] Interestingly, there is a 2% regression when running the Fibonacci benchmark with data-flow optimization, but without inlining. As our optimizations do not perform any transformations in this case, and this regression is not reproducible in isolation, we assume that this is a side-effect of different memory layout caused by optimizations in other functions.

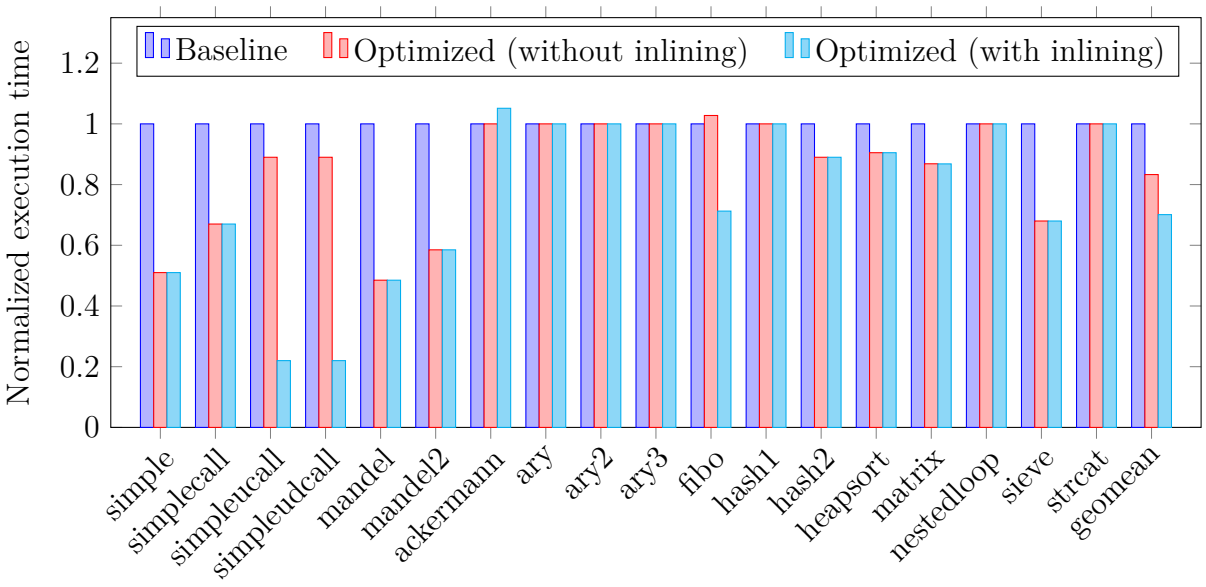


Figure 5.1.: Normalized execution times for standard PHP microbenchmarks for baseline (blue), data-flow optimizations without inlining (red) and with inlining (cyan). Lower is better. The last column is the geometric mean.

time, starting from a baseline with inlining enabled, but data-flow optimizations disabled. For the tested benchmarks only dead code elimination, copy propagation, assignment contraction and type specialization had a non-trivial impact.

The contributions of the individual optimizations do not always sum to one. For the mandel and sieve benchmarks the reason is that assignment contraction changes `ASSIGN_ADD $a, $b` operations into `$a = ADD $a, $b`, which then allows type specialization to act on them. For the simpleu(d)call benchmarks this is not the case. Here, the elimination of dead code does not affect the number of specialized instructions. Rather, it appears that the use of specialized instructions has a larger overall impact if there are fewer non-specialized instructions interspersed.

Similarly, there are a number of cases where the individual optimizations sum to more than one. The reason for this is that many optimizations have some degree of overlap, e.g. copy propagation and assignment contraction may lead to the same effective result in some cases. The most extreme example is the matrix benchmark, where all four optimizations contribute individually, but assignment contraction subsumes everything else.

The second set of microbenchmarks shipping with PHP (micro_bench.php) are different in nature: They benchmark the repeated execution of a single operation, or a combination of few operations. As such, these benchmarks are of very little value to us, as they essentially only test whether we can DCE a particular operation. We are able to DCE the loop body in 9 out of 34 cases. In the remaining cases we fail to show that the operation can never generate a run-time warning.

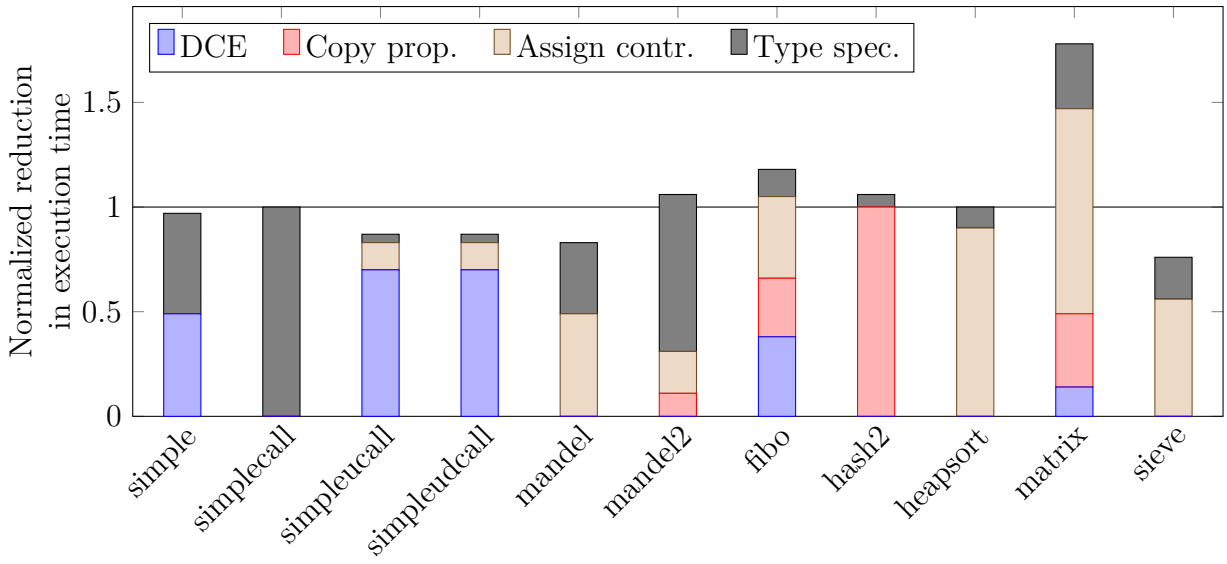


Figure 5.2.: Effect of individual optimizations on microbenchmarks. The black line represents the performance improvement if all optimizations are enabled, while the stacked bar charts show the impact of individual optimizations. The individual parts do not necessarily sum to one, if one optimization improves another or if there is overlap between different optimizations. Inlining is enabled in all cases.

5.2. Real applications

Of course, the ultimate goal of this work was to improve performance of real-world applications, rather than of microbenchmarks.

We have tested two applications to estimate real-world impact: The first is WordPress, which is a popular blogging platform and without doubt the most widely deployed application written in PHP. The second is MediaWiki, which is the software powering Wikipedia.

For both applications we have measured response times in sequential execution.[2] For WordPress the homepage was tested, while for MediaWiki the Wikipedia page of Barack Obama was used (which is also used by the Wikimedia Foundation for benchmarking).

For WordPress we observe an improvement of 1% in execution time, and for MediaWiki a slightly larger improvement of 2%. In both cases additionally enabling inlining leads to a minor positive change (<0.5%) if only few small functions with mostly constant arguments are inlined. If inlining is used aggressively there is a significant negative impact on performance. E.g. for MediaWiki execution time increases by 5% if all eligible calls are inlined to one level.

Due to the small differences in execution time, it is hard to quantify the run-time impact of individual optimizations. However, based on statically collected statistics, it is

[2]The noise when measuring throughput under high concurrency in a virtualized environment (using Facebook's oss-performance tool) was too high to resolve small timing differences.

likely that assignment contraction is the primary contributor. For WordPress 3.7% of all instructions are affected by assignment contraction, while for MediaWiki it is 4.3%.

The effect of all other optimizations is at least an order of magnitude smaller: The combination of SCCP and DCE removes 0.5% of instructions in WordPress and 0.2% in MediaWiki. Similarly, copy propagation removes 0.2% in both cases.

However, in this case the more interesting question is not which optimizations apply, but rather why so few of them do. While a 1-2% improvement in response times of realistic applications is in line with our initial expectations, we did harbor a hope of seeing better results.

One way to gain insight into this question is by considering the amount of type information that is available for these applications: In both cases nearly half of all SSA variables have type *any*, of which 70% are additionally marked as potential references. As reference variables can never be optimized and there is little that can be done with type *any* variables, this means that half of all variables are effectively excluded from optimization. A key contribution to the number of potential reference variables is the fact that the current compiler has to assume that all functions defined outside the current file use by-reference argument passing. If this limitation could be removed (which is likely in the longer term) more type information would become available.

Additionally, typical PHP applications do not tend to contain the kind of computationally heavy, tightly looped code that can be effectively optimized using type specialization. Rather, PHP applications commonly operate on strings and arrays, where specialization is less effective as the involved operations are intrinsically more expensive, so that overhead of type-checking plays a lesser role.

It is not very surprising that optimizations like SCCP and DCE are not very effective on non-inlined code, as programmers do not tend to intentionally write large amounts of dead code. If we use a relatively aggressive inlining heuristic, which inlines all eligible calls to one level, the amount of instructions removed by a combination of SCCP and DCE increases to 6% (from previously 0.5%). This suggests that if we can counter the other disadvantages of inlining discussed in section 4.8, there is a significant potential for optimization from this avenue. As an alternative to inlining one could also consider creating clones of functions for certain constant arguments. This still allows SCCP and DCE to be effective, but does not cause issues with increased variable space and also preserves stack traces.

6. Conclusion and Outlook

The goal of this thesis was to establish whether it is possible to improve the performance of PHP, by improving the quality of the generated bytecode through the use of classic data-flow optimization techniques. The primary challenge in doing so, is that these techniques have been developed in the context of statically typed and compiled languages, while PHP is dynamically and weakly typed and additionally features a plethora of other dynamic language features.

We operate on static single assignment form and use type inference to determine the possible types of variables. Based on this, we perform a number of different optimizations, including constant propagation, dead code elimination and copy propagation, as well as more PHP-specific transformations such as type specialization and assignment contraction.

We have found that significant performance gains are possible for numerically intensive and tightly looped code, as is typically found in benchmark scripts. We achieve a mean speedup of 1.42× on PHP's own benchmark suit. However, when considering real applications we have found the speedup to be limited to 1-2%. For real applications, the most effective optimization is assignment contraction, which is a specific form of type specialization. There are a number of reasons why we do not see better results, which also suggest potential avenues for further research:

First, we currently suffer from a limitation of the PHP opcode caching layer, which requires each file to be compiled independently. As this prevents us from knowing the signatures of functions defined in different files, we have to pessimistically assume use of by-reference argument passing. As potential reference variables are excluded from optimization, this significantly reduces the scope of applicability. If this limitation can be removed in the future, our approach should show better results without further modifications.

Second, our implementation of type inference currently operates on individual procedures only. This is problematic because at present many popular projects do not yet specify type information in function signatures (or only type information that is hard to optimize on). As such, there is only relatively little known type information which can be used for inference. This situation could be improved by using inter-procedural type inference. However, the degree to which inter-procedural inference is possible is again limited by the previous point.

Third, optimizations such as constant propagation and dead code elimination only work effectively if inlining is used. However, inlining comes at the cost of increased program size and a larger variable space. One way to ameliorate the latter problem would be to implement a register allocator for compiled variables. Another interesting approach would be to forgo inlining and instead clone functions for specific constant arguments or types.

This would allow constant propagation and dead code elimination to become effective, while avoiding some of the inlining overhead and additionally preserving stack traces. However, it is unclear whether the additional optimizations would still be overshadowed by the program size increase.

Fourth, while we have implemented a number of different optimization passes, many more compiler optimizations are known. For example, we did not include any mechanism for redundancy elimination, such as global value numbering or partial redundancy elimination. Similarly, we did not attempt to implement any loop optimizations, which are known to be particularly effective.

Finally, this thesis has been focused on purely static optimizations. However, this disregards an important opportunity that is available in interpreted languages, namely the possibility of collecting type information at runtime. This would allow the insertion of guards for likely input types and optimization based on them (although a version of the function without type restrictions would have to be retained).

To conclude, the data-flow optimization framework in its current state only provides a relatively small performance improvement for real-world applications. However, it lays an important foundation for further optimization work in PHP, some possible directions for which have been outlined above. Additionally, we believe that this approach to optimization will only become more applicable in the future, as both the PHP language and community increasingly embrace the use of type annotations. Finally, it is likely that the implemented optimizations will also prove useful when the PHP project revisits the possibility of using a just-in-time compiler.

A. Source Code

The source code for this thesis is available at https://github.com/nikic/php-src/tree/opt, though some parts are already contributed upstream. As this branch is still under development, https://github.com/nikic/php-src/tree/opt_orig has been retained as the state at the time of this writing.

All-in-all the implementation amounts to approximately ten thousand lines of code, though a significant part is made up of experiments not described in this thesis. The following is a very brief overview of structure and options. Familiarity with the PHP implementation is assumed here, this is not an end-user description.

The optimization component of opcache is located in the ext/opcache/Optimizer directory. The bulk of our work is located in the ssa/ subdirectory. Additionally ssa_pass.c includes overall management, statistics.c collects statistics and inlining.c performs inlining.

SSA optimizations are registered as pass 6, while inlining is pass 8. As shortcuts, the optimization level `0xf1f` disables data-flow optimization, `0xf3f` enables it and `0xf9f` additionally enabled inlining (opcache.optimization_level).

Individual data-flow optimization passes can be enabled/disabled using opcache.ssa_opt_level. Debug dumps are available through opcache.ssa_debug_level and can be restricted to one function using opcache.ssa_debug_func.

opcache.opt_statistics=1 will dump statistics collected during optimization.

opcache.opt_statistics=2 will dump a trace of all instructions that were compiled, along with inferred type information. We provide an analyze.php program in the root directory, which can be used to filter this trace for specific types and instructions and present aggregated results.

opcache.opt_statistics=3 will instrument the bytecode with type checking instructions. The purpose of this mode is to verify type inference results.

Our test porting utility is portTests.php in the root directory, however the ported tests are already committed in this branch.

Acronyms

AST Abstract Syntax Tree

CFG Control Flow Graph

DCE Dead Code Elimination

DF Dominance Frontier

DFS Depth First Search

HHVM HipHop Virtual Machine

IR Intermediate Representation

JIT Just-In-Time compiler

RPO Reverse Post-Order

SCCP Sparse Conditional Constant Propagation

SHM SHared Memory

SSA Static Single Assignment form

VM Virtual Machine

Bibliography

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014. ACM.

[2] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26, Berlin, Heidelberg, 1995. Springer-Verlag.

[3] Frances E. Allen and John Cocke. Graph-theoretic constructs for program control flow analysis. Technical report, Technical Report RC-3923, IBM Research, 1972.

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

[5] Paul Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP.* PhD thesis, Trinity College Dublin, 2009.

[6] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. *SIGPLAN Not.*, 35(5):321–333, May 2000.

[7] 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 *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 35–44, New York, NY, USA, 2008. ACM.

[9] F. Bradner and D. Novillo. Propagating information using SSA. In *Static Single Assignment Book*, chapter 8. June 2015.

[10] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881, July 1998.

[11] P. Brisk. Properties and flavours. In *Static Single Assignment Book*, chapter 2. June 2015.

[12] Stefan Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 429–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Brett Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005.

[14] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 150–164, New York, NY, USA, 1990. ACM.

[15] 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 *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, pages 253–267, Berlin, Heidelberg, 1996. Springer-Verlag.

[16] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Softw. Pract. Exper.*, 4:1–10, 2001.

[17] 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, Oct. 1991.

[18] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[19] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 283–300, New York, NY, USA, 2009. ACM.

[20] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[21] Loukas Georgiadis, Robert E. Tarjan, and Renato F. Werneck. Finding dominators in practice. *J. Graph Algorithms Appl.*, 10(1):69–94, 2006.

[22] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[23] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, Jan. 1979.

[25] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.

[26] Nikita Popov. Forbid binding methods to incompatible $this. PHP internals mailing list, Mar. 2016. http://markmail.org/message/zepnhdyr3kij6di6 (retrieved July 1, 2016).

[27] Nikita Popov. Forbid dynamic calls to scope introspection functions. PHP RFC proposal, May 2016. https://wiki.php.net/rfc/forbid_dynamic_scope_introspection (retrieved July 1, 2016).

[28] Andrei Rimsa, Marcelo d'Amorim, and Fernando Magno Quintão Pereira. Tainted flow analysis on e-SSA-form programs. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 124–143, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[30] Dmitry Stogov. Zend JIT open sourced. PHP internals mailing list, Feb. 2015. http://markmail.org/message/qwufjtskwzulz4eq (retrieved July 1, 2016).

[31] Michiaki Tatsubori, Akihiko Tozawa, Toyotaro Suzumura, Scott Trent, and Tamiya Onodera. Evaluation of a just-in-time compiler retrofitted for php. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 121–132, New York, NY, USA, 2010. ACM.

[32] Joe Watkins and Phil Sturgeon. Typed properties. PHP RFC proposal, Mar. 2016. https://wiki.php.net/rfc/typed-properties (retrieved July 1, 2016).

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

[34] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 278–287, New York, NY, USA, 2010. ACM.

[35] Michael Wolfe. J+=J. *SIGPLAN Not.*, 29(7):51–53, July 1994.

[36] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[37] 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 *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 575–586, New York, NY, USA, 2012. ACM.

[38] Peachpie PHP compiler to .NET, 2016. https://github.com/iolevel/peachpie (retrieved July 1, 2016).

[39] Quercus: PHP in Java. http://quercus.caucho.com/quercus-3.1/doc/quercus.xtp (retrieved July 1, 2016).