ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A PHP Implementation Built on GraalVM

Bachelor Thesis

Andrin Bertschi

Monday 14th September, 2020

Advisors: Prof. Dr. Zhendong Su, Dr. Manuel Rigger
Department of Computer Science, ETH Zürich

Abstract

PHP is a popular, weakly typed, general purpose programming language. Originally designed for building dynamic web pages, the language has since gained wide adoption in server-side web development. In this work, we describe the design and implementation of *graalphp*, an experimental compiler and runtime for PHP hosted on Truffle and GraalVM. GraalVM is a virtual machine that supports execution of multiple languages, which are implemented as Abstract Syntax Tree (AST) interpreters based on Truffle. GraalVM uses Graal as its JIT compiler to compile frequently executed code fragments to machine code. We implement a subset of the PHP language to run synthetic benchmarks by *The Computer Language Benchmarks Game*. We compare peak performance of our implementation against PHP 7 as well as alternative implementations such as HHVM, JPHP and an early alpha version of PHP 8. Experimental results indicate that our runtime reaches competitive results with performance gains of up to 859% compared to PHP 7. These preliminary results suggest that a Truffle-hosted PHP implementation might be significantly faster than existing language implementations.

Contents

Chapter 1

Introduction

Dynamically typed programming languages are well regarded choices for writing websites. Their expressiveness and fast build-compile-run cycle allow for rapid prototyping, debugging and testing. Despite their popularity, dynamical languages impose challenges on a fast language implementation. Type checking is delayed until runtime because their types might change during execution. PHP is one of these languages. Its reference implementation, *Zend Engine*, implements PHP with a bytecode interpreter without a dynamic compiler.

In order to accelerate performance of PHP, previous work translated PHP code to a statically typed language such as C++. Examples of this approach include *phc* [1] and *HPHPc* [2]. Their method relies on ahead-of-time optimizations of mature existing compilers but is challenged to implement dynamic features such as *eval*[1], which evaluates source code at runtime. Another technique to improve performance is dynamic compilation. *HHVM* [3, 4] is an example of a dynamic compiler built from ground up to execute Hack[2]. During execution, a dynamic compiler gathers profiling feedback and translates frequently executed code fragments to machine code.

GraalVM is a virtual machine that supports the execution of multiple languages, which are implemented as Abstract Syntax Tree (AST) interpreters based on Truffle. Truffle is a language implementation framework. GraalVM uses Truffle and a dynamic compiler called Graal to automatically derive aggressive compiler optimizations to just-in-time (JIT) compile code fragments.

Truffle-hosted language runtimes for dynamic languages have shown competitive performances [5]. Furthermore, TIOBE and PYPL [6, 7] rate PHP within the 10 most popular languages and other related rankings show sim-

[1] Eval: `https://www.php.net/manual/en/function.eval.php`, *archived: url*

[2] Hack is a dialect of PHP: `https://hacklang.org/`, *archived: url*

ilar results [8, 9]. Although PHP is a popular language, no research has been done on a GraalVM-hosted implementation. Additionally, we believe that many existing language implementations do not aggressively apply dynamic optimizations. This makes PHP still an interesting target for optimizations.

In this thesis, we present *graalphp*, an experimental compiler for PHP 7+ hosted on Truffle and GraalVM. We use Truffle to model PHP programs as an AST interpreter which can further be optimized and JIT compiled by Graal on GraalVM. We highlight the design of important language features to implement a subset of PHP. These features were dictated by synthetic benchmarks. In our evaluation, we perform a preliminary performance comparison of potential speedups against the most maintained implementations of PHP. Given that our implementation is not yet fully realized, we hope to lay foundations for future work towards a feature complete implementation.

1.1 Contributions and Scope

We make following contributions.

- We develop a language implementation for PHP, *graalphp*, based on Truffle and GraalVM. We thereby describe key design aspects and publish our work as open-source software [10].
- We run our language implementation on three benchmarks by *the Computer Language Benchmarks Game* [11]. We compare peak performance against PHP 7 and alternative implementations such as HHVM, JPHP, and PHP 8 Alpha.
- We modularize an existing parser for PHP and provide it as a standalone parsing library for Java.

Explicit non-goals are completeness with respect to the language specification. PHP is an expressive language with a variety of features [12]. Completeness exceeds the time margin of this Bachelor thesis. Nevertheless, our preliminary results indicate competitive peak performance. The benchmark results suggest that a Truffle-hosted implemented for PHP might be significantly faster than existing implementations.

1.2 Structure of this Document

In Chapter 2 we present background information on PHP, GraalVM, AST interpreters and Truffle. In Chapter 3 we elaborate on designing and implementing *graalphp*. Next, in Chapter 4 we evaluate peak performance of our language implementation and compare results. We benchmark *graalphp*

against the reference implementation as well as alternative runtimes. Subsequently, Chapter 5 presents related work while Chapter 6 elaborates on future work. Finally, we conclude our findings in Chapter 7.

Chapter 2

Background

In this chapter we clarify important topics to prepare a reader for the document. We introduce background information on PHP, Graal, GraalVM, AST interpreters and Truffle.

2.1 PHP Programming Language

The PHP Hypertext Processor (PHP) is a weakly typed, dynamic programming language. The language was first released in 1995 as an imperative scripting language for server side web development [13]. The current version as of 2020 is PHP 7.4. Modern PHP includes features such as object-oriented programming, traits, closures, reflection, null coalescing[1] and array destruction[2]. Only in 2014 was PHP formally defined by a language specification, which was derived from the behavior of its reference implementation *Zend Engine* [12, 14]. Listing 1 portrays a simple example of a PHP program.

```
1 <?php
2 function hello($who) {
3     return "Hello " . $who;
4 }
5 echo hello("PHP") . "\n";
```

Listing 1: Sample PHP code.

Alike many scripting languages such as JavaScript and Python, PHP is dynamically typed. In dynamic typing, type checks occur at runtime. The function `hello($who)` in Listing 1 does not specify a type and may be called with a string `"PHP"` or an integer `1337`. Both calls are valid invocations.

[1] Null coalescing: `$value = $input ?? "fallback, input not defined";`
[2] Array destruction: `[$a, $b] = get_results();`

Advantages of dynamically typed languages include flexibility and fast prototyping. Alternatively, statically typed languages are type checked at compile type. This may improve program structure and can help catching logic errors earlier in program development [15].

PHP is further a weakly typed language. Opposite to strongly typed, a weakly typed language allows to mix different types. In PHP, we may add a string and an integer, i.e. `"1336" + 1`, and receive an integer. PHP 7 introduced a stricter, opt-in type system which bails out if type mismatches occur [16]. The type system allows for type declarations in function arguments, return values and class properties, and can be enabled with a compiler directive on a file basis. However, implementation of type declarations is outside of the scope of this thesis.

Previous work implies that dynamic languages are often implemented with interpreters, e.g. [17, 18, 19, 20]. This is due to the nature of dynamic types. Type checking must be delayed until types are known which is at runtime. We will introduce interpreters to a greater extent in Section 2.4.

2.2 Graal and GraalVM

With Java 9, the Java based JVM compiler interface (JVMCI) was introduced to the Java Platform. JVMCI allows a dynamic compiler written in Java to be used by the JVM [21]. A dynamic compiler collects profiling data to understand program execution. It can apply dynamic optimization techniques to optimize frequently executed code. Compilation techniques are typically performed at build time, so-called *Ahead-of-Time* (AOT) or at execution time, so-called *Just-in-Time* (JIT).

Dynamic compilers perform JIT compilation. They are often written in low-level programming languages such as C++. Among many, this is the case for V8 (JavaScript) [22], HotSpot (Java) [23] and PyPy (Python) [24]. Writing a JIT compiler in a managed language comes with a number of benefits including memory management, faster prototyping, portability, and tooling support [20, 21].

Graal is a dynamic compiler written in Java which leverages these benefits. It implements the JVMCI interface and can be loaded at runtime similar to Java Agents. Starting with JDK 10 (Java Development Kit), Graal ships as part of the OpenJDK distribution as an experimental JIT compiler. Furthermore, Graal is part of GraalVM, a distribution of Java, which uses Graal as its default JIT compiler [25]. We will use GraalVM as the Java distribution for *graalphp*. Graal makes use of speculations to apply aggressive optimizations. Among many optimizations, it can cut off cold branches from compilation and bail out to its interpreter when assumptions no longer hold. More detailed information about Graal and its graph based intermediate

representation (IR) can be found in [26, 27]. In Section 2.5, we will summarize some optimization techniques used in conjunction with Truffle to implement a guest language on GraalVM.

2.2.1 GraalVM Native Image

GraalVM Native image allows AOT compilation of Java code to a standalone executable. Compared to a conventional Java virtual machine, a native build has lower memory overhead and faster startup time [28]. We will provide a native build of our language implementation, *graalphp-native*, but will refer to further work for optimizations on the native build. A native build will indicate some of the potential of AOT compilation for a Truffle-hosted implementation of PHP.

2.3 Abstract Syntax Trees

An *Abstract Syntax Tree* (AST) is a tree representation of source code. Each node constitutes a language construct. An AST is abstract in the sense that syntactic details are omitted and only semantic elements are emphasized. Figure 2.1 depicts how an assignment $\$a = 668 * 2 + 1$ is translated into an AST. We will use this example throughout the chapter.

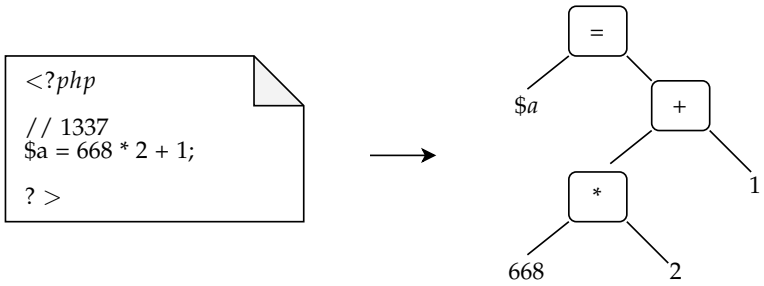

Figure 2.1: Example of a source to AST derivation.

Truffle-hosted languages are implemented as AST's and interpreted by Truffle before they are partially evaluated and compiled to machine code. In the next sections, we will introduce AST interpreters and Truffle to a greater extent.

2.4 AST Interpreters

When implementing a new programming language, an intuitive approach towards an implementation is the *Interpreter Pattern* [29]. An AST is derived from source code and for each operation, literal and variable in the program

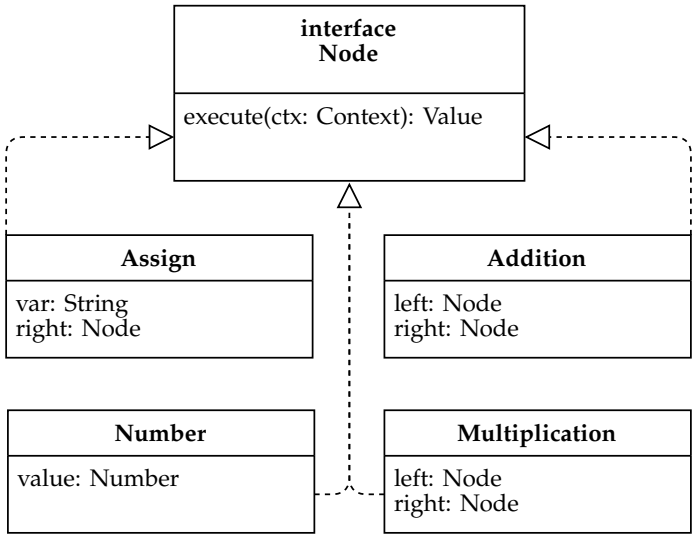

Figure 2.2: Example of AST interpreter pattern.

an interpreter node is created. The base node typically defines an abstract contract such as $execute(ctx : Context) : Value$ which all children implement. To execute the program, the root node is called which then recursively calls execute on all children in post-order traversal. Continuing the example of $a = 668 * 2 + 1$, we may build an AST interpreter visualized in Figure 2.2 and instantiated below:

```
1  // assign 668 * 2 + 1 to variable a
2  assign = Assign("a", Add(Mul(Num(668), Num(2)), Num(1)));
3  assign.execute(ctx: Context);
4  /*
5    Call Graph:
6    Assign.execute()
7     \__Add.execute()
8         \__Mult.execute()
9         |     \__Num.execute()
10        |     |__Num.execute()
11        |___Num.execute()
12 */
```

Listing 2: AST interpreter method dispatch.

Note the call graph in Listing 2. In order to get the result value of 1337, we need to dispatch all *execute* methods. The interpreter pattern is conceptually simple to implement. Nevertheless, parsing and tree walking on each run causes overhead. Most importantly, dispatch on all *execute* methods cause

overhead. Therefore, execution speed is often moderate compared to native compilation [19]. One approach to decrease overhead is to introduce an intermediate representation (IR). Instead of parsing $\$a = 668 * 2 + 1$ on each run we may introduce assembly like instructions with a fixed number of operands. This IR is called bytecode in Java and opcode in PHP. Compilation is now two folded. First we compile to the IR, and when executing the program, we interpret the IR. This gives opportunity to apply initial optimizations ahead-of-time (AOT). As an example, constant folding will simplify our snippet $\$a = 668 * 2 + 1$ to 1337. Virtual machines have a multitude of techniques reaching from proof-based optimizations such as constant folding to flow sensitive rewrites such as dead code elimination [30]. PHP's reference implementation[3] accelerates execution with opcode caching. Extensions such as APC[4] and Opcache[5] store AOT compiled bytecode in shared memory. This removes the first fold of reading the script and applying initial optimizations across requests. Despite these efforts, PHP's reference implementation does not include a dynamic compiler in its current version[6].

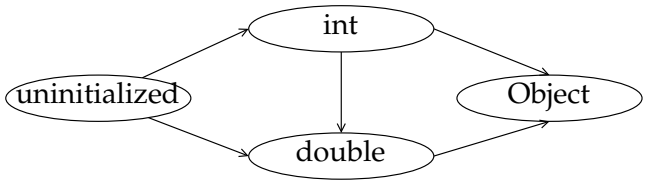

Figure 2.3: Truffle node specializations for addition in PHP.

2.5 Truffle

Truffle is a language implementation framework for Graal. It allows us to implement a new programming language by writing an AST interpreter. The interpreter executes all children nodes and performs specializations. In other words, it adds profiling and type information to the executed program [32]. In dynamic languages, we often only know at runtime what concrete types occur at a given code segment. Based on these concrete types, a Truffle-hosted AST can rewrite itself and prefer an node implementation which matches a given type. Graal then automatically derives optimizations from the interpreter using partial evaluation and profiling feedback [33, 34]. This follows the principle of "do-not-repeat-yourself" as we model an AST

[3] In this thesis referred to as Zend Engine or Zend

[4] APC: `https://www.php.net/manual/en/book.apc.php` *archived: url*

[5] Opcache, list of AOT optimizations: `https://www.php.net/manual/en/opcache.configuration.php`, *archived: url*

[6] Version 7.4 as of writing this thesis, a dynamic compiler is planned for PHP 8 [31]

interpreter and rely on mature infrastructure of Truffle and Graal to further optimize the executed program.

Specialization and Partial Evaluation

Specializations address optimization issues found with dynamic typing. As an example, we may write an addition node as shown in Listing 4 (details omitted for simplicity). Truffle provides a DSL (Domain Specific Language) with an annotation processor to generates wrapper classes. For instance, in Listing 4, we implement the first node specialization as follows:

```
1 @Specialization(rewriteOn = ArithmeticException.class)
2 protected long add(long left, long right) {
3   return Math.addExact(left, right);
4 }
```

This specialization assumes that the node children return *long* datatypes. The annotation processor generates a state machine with states for all specialization. In our case, there are four states: *Uninitialized*, *long*, *double*, and *Object*. The states form a lattice and transition from an uninitialized variant, to primitive data types to generic Java Objects in a constant number of steps (Figure 2.3). The use of primitive datatypes avoids boxing which reduces overhead, while a generic specialization uses Java *Object*'s to handle all cases. Graal assumes that node specializations are stable (i.e. node rewriting has stopped, and all node children are treated as Java *final*) and aggressively performs inlining and constant propagation of nodes under the same tree. This removes the dispatch of the *execute* methods between nodes and allows Graal to emit efficient machine code. Profiling feedback from Truffle ensures that partial evaluation does not explode in code size and only includes specializations which were used during execution in the interpreter. This is only speculative, however. We speculate that an addition, for instance, is only performed on *long* types and not all other types which addition in PHP also supports. If these assumptions no longer hold, Graal can replace stack frames of the compiled code, discard the emitted machine code, and transfer execution back to the AST interpreter, where we gather new profiling feedback. This process is referred to as *deoptimization* [32, 34].

The example in Listing 3 emphasizes on the idea of partial evaluation. Nodes of the AST are considered as final and then inlined which removes the method dispatch.

```
1 // 668 * 2 + 1
2 Add(Mul(Num(668), Num(2)), Num(1)).execute(ctx);

3 // 1.
4 Mul(Num(668), Num(2).execute(ctx)) + Num(1).execute(ctx);
```

```
// 2.
Num(668).execute(ctx) * Num(2).execute(ctx) + Num(1).execute(ctx);

// 3.
668 * 2 + 1;
```

Listing 3: Introduction to partial evaluation. For the sake of example, numbers are constant values.

```
@NodeInfo(shortName = "+")
public abstract class PhpAddNode extends PhpBinaryNode {
  // children nodes declared here ...

  @Specialization(rewriteOn = ArithmeticException.class)
  protected long add(long left, long right) {
    return Math.addExact(left, right);
  }
  @Specialization
  protected double add(double left, double right) {
    return left + right;
  }
  @Specialization
  Object executeGeneric(Object left, Object right) {
  // ...
  }
}
```

Listing 4: Truffle addition node to add PHP values.

Chapter 3

Design & Implementation

In this chapter we highlight the design and implementation of *graalphp* and how we modeled important language features. We first establish different design approaches towards an implementation. Thereby, we introduce an opcode and a source-based approach to leverage *Zend Engine's* existing internal representation and to parse plain source code. We will then elaborate on the architecture of our Truffle-hosted AST interpreter. Subsequently, the main part of this chapter is dedicated to the design of important language features. These features enable us to perform a peak performance comparison with *Zend Engine* and alternative runtimes. At the time of writing this report, our core language runtime reached around 4000 lines of code.

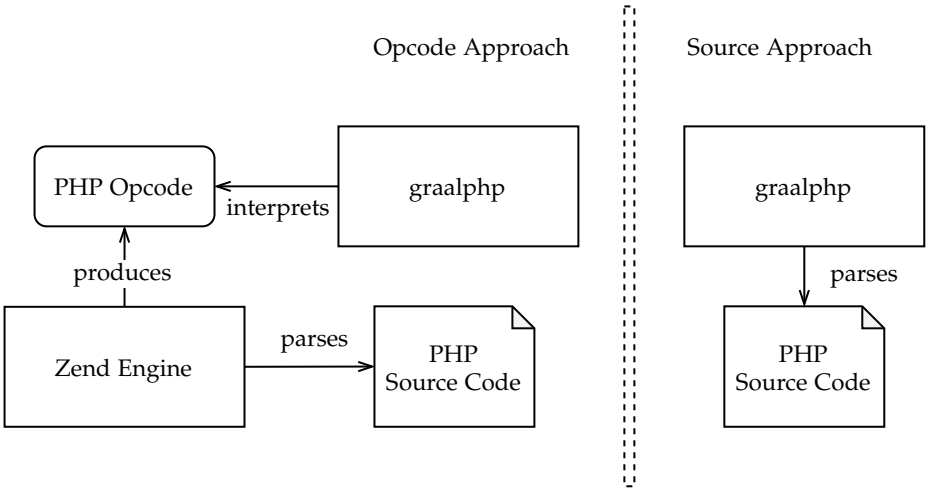

Figure 3.1: Design evaluation opcode and source approach.

3.1 Design Evaluation

We distinguish between two layers of abstraction when implementing a GraalVM-hosted compiler for PHP. The first layer integrates into the opcode format of *Zend Engine* – PHP's reference implementation, while the second layer parses plain source code. For more detail on *Zend Engine*'s internal representation, we refer to Related Work in Section 5.1.

Opcode Approach *Zend Engine* is a bytecode interpreter. There are different ways to access *Zend Engine*'s internal bytecode representation. An opcode-based method leverages Zend's existing parsing and optimization stack. The VM exposes an extension API to analyze and modify its internal representation. This is for instance used by parsing accelerators such as *APC*[1] which skips parsing of the same source file. Our first design suggestion is based on an Zend opcode interpreter for Truffle. Advantages of this method include reusability of the parsing infrastructure. Even for upcoming syntax changes in PHP this approach is always compatible with the latest source version, supposing no new opcodes are introduced. Zend's opcodes are untyped which may impose an overhead on type checking. Furthermore, alike many bytecode interpreters, Zend represents control flow with implicit jumps. This requires manual reconstruction of high level control flow in order to effectively leverage Truffle's optimization stack. Work done by Mosaner et al. [35] evaluated loop reconstruction and extraction for on-stack replacement in LLVM bitcode. They managed to partially reconstruct high-level constructs for a Truffle-based interpreter. An opcode interpreter approach would therefore follow a similar method. The illustration in Figure 3.1 on the left side depicts this approach.

Source Approach A source-based interpreter approach parses plain-text PHP code. It then manually constructs an AST and translates it to match Truffle's implementation contracts. This method requires a sophisticated parsing library to support PHP's diverse range of language concepts. We can either use an existing parser or write a new parser from scratch. A source-based interpreter is affected by syntactic language changes. New versions of PHP are typically released every year and receive support for a duration of two years [36]. In Appendix A.1, we will evaluate parsing libraries for PHP. The illustration in Figure 3.1 on the right side depicts this approach.

While a method with an opcode interpreter is less prone towards new language features, we believe a source-based interpreter has more potential for optimizations. This is because we do not lose semantic information and can

[1] opcode cache: `https://www.php.net/manual/en/book.apc.php` *archived: url*

obtain the source in its original textual representation. Hence, we propose a design based on a source interpreter.

3.2 Truffle-hosted Source Interpreter

We propose a design based on PHP's source representation. In Figure 3.2 we visualize the designed architecture.

Our implementation consists of two main components, *graalphp-language* – the Truffle language implementation, and *graalphp-parser* – a PHP source code parser. Upon executing PHP we parse a source file and create an abstract syntax tree. We will then execute the AST until no more node specializations occur. This will yield profiling feedback for Graal to partially evaluate and compile frequently executed code fragments to machine code.

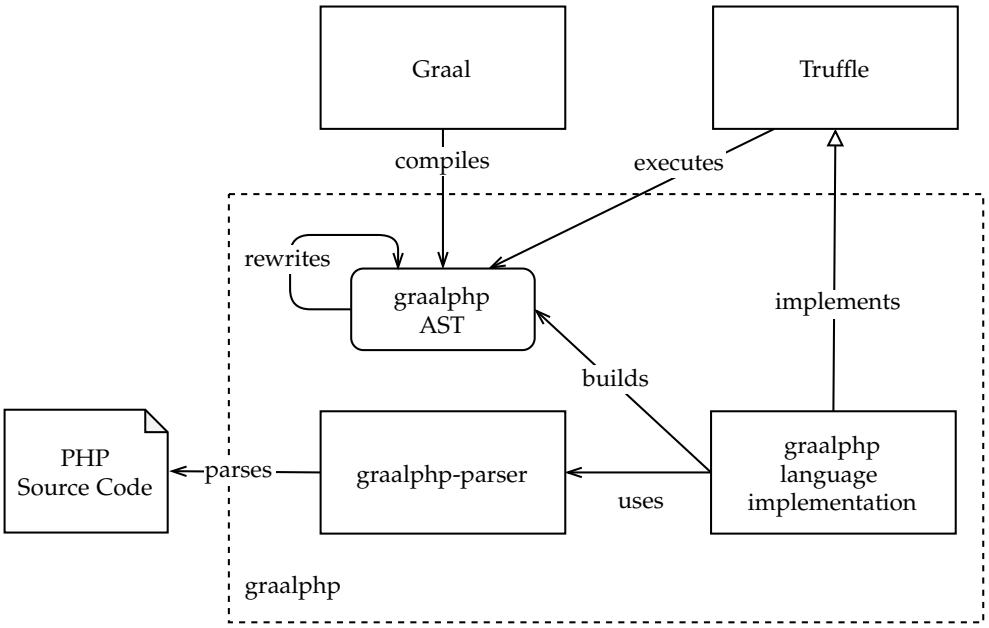

Figure 3.2: Design overview of graalphp.

The next sections are dedicated to how we parse source code and build an abstract syntax tree. Subsequently, we portray design decisions and implemented language features and conclude with remaining work.

3.2.1 Parsing Source Code

We evaluated existing Java based parsers for PHP but were unsatisfied with their features and speed. They either did not support PHP 7+, did not

scale well for parsing large source files, did not provide a visitor pattern for AST traversal, or were tightly integrated into other products and thus not reusable as a library. We will elaborate on our analysis and contribution to a parser in Appendix A.1. In brief, we forked the Eclipse PDT Tools [37] and modularized their parser to be usable as a library. Our fork is published as *graalphp-parser*, a standalone parsing library with competitive parsing speeds.

3.2.2 AST Nodes

Having a parsing library with tree traversal in place, we can walk through the source AST in one take and create PHP nodes for Truffle. While our class hierarchy is diverse, it can be simplified to the hierarchy visualized in Figure 3.3. The concepts introduced throughout this chapter are based on statements, which do not evaluate to a value, and expressions, which evaluate to a result value. Unlike other programming languages, PHP does not restrict execution start to a *main* function but starts executing the first statement at the top level of a script. We therefore distinguish between two different call targets[2]. A target derived from a *PHP Global Root Node*, which contains all statements at the top level of a script, and a *PHP Function Root Node*, which represents user functions.

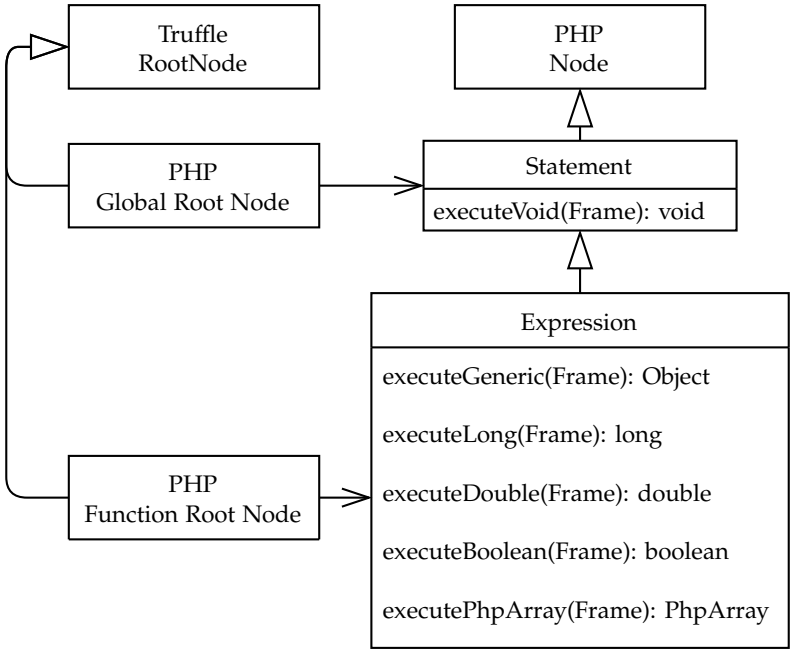

Figure 3.3: PHP root classes to model statements and expressions.

[2] A call target represents a tree hierarchy of nodes to call

We see in Figure 3.3 that *Statement* and *Expression* are common parent nodes in our hierarchy. They define an abstract contract `executeVoid(Frame): void` and `executeGeneric(Frame): Object` which all children implement. The other `execute*(Frame)` methods are derivations of these generic cases. They are auxiliary methods and specialize on different return types, for instance *long*. We will further introduce our typing system in the next section.

3.3 Modeling Language Features

PHP is an expressive language with a variety of features. In order to show a preliminary performance comparison, we focus on language features imposed by our synthetic benchmarks (see Section 4.1.2). We will now give an overview in Table 3.1 before we advance to design and implementation decisions of these features.

Language Feature	**Comment**
Data Types	*int*, *float*, *double*, *array*
Functions and Variables	function declaration, invocations and variables
Control Flow	*If/else*, *while*, *do-while*, *for*, *return*, *break*, *continue*
Arrays	create, read and writes arrays, reference operator
Auxiliary operations	arithmetical operations (i.e. $+$, $-$, $*$, $/$, $++$, $--$, $<<$, $>>$), logical operations (i.e. !, &&, $\|\|$), relational operations (i.e. $>$, $<$, $>=$, $<=$, $==$, $!=$)

Table 3.1: Overview language features.

3.3.1 Data Types

In order to meet functionality imposed by our synthetic benchmark, we implemented language semantics for *int*, *boolean*, *float* and *array* [38]. *Array*'s are aggregations of the previously mentioned datatypes, including arrays themselves. In Table 3.2 we summarize their characteristics.

PHP Type	Detail	Mapping in Java
bool	TRUE or FALSE	boolean
int	at least 4 bytes, signed	long
float	at least IEEE 754 64-bit double-precision	double
array	generic container	-

Table 3.2: Implemented data types and their mapping to Java data types.

PHP's language specification defines integers as 4 bytes but leaves an upper bound as an implementation detail [12, Section 05-types, The Integer Type]. Furthermore, integer must be converted to float on overflow.

In order to reduce rounding issues for integers larger than 4 bytes, we model integers as 8 bytes Java *long*. We model overflow with explicit Truffle node rewrites in all nodes which modify a long value. By giving long specializations higher priority than double, we can utilize Truffle's *@ImplicitCast* and convert long values to match double specializations.

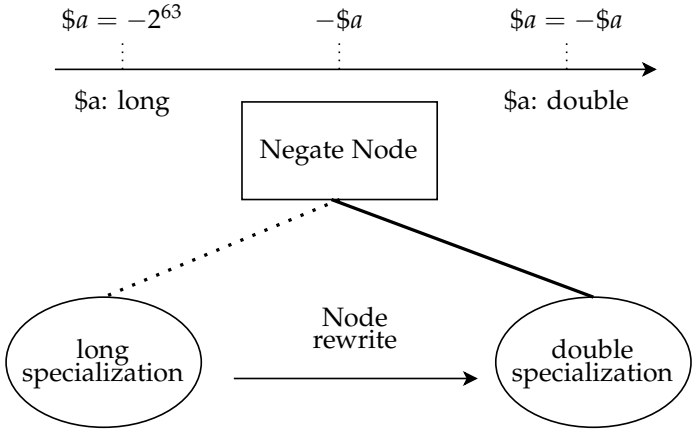

Figure 3.4: Overflow of long data type, implicit cast to double.

This concept is illustrated in Figure 3.4. In this example, we assign -2^{63} to variable $\$a$ and negate its value. -2^{63} is the smallest value we can store in Java long. Negating does overflow the domain of long. While we execute the operation, we instruct Truffle to rewrite the *Negate Node* to use a double specialization instead. We then implicitly convert long to double and continue negating in the double specialization. The resulting type of $\$a$ after the assignment is a Java double.

While bool and float follow similar approaches, arrays cannot be modeled

with primitive data types. We will elaborate on our array design in a forthcoming section.

3.3.2 Functions

Recall that we introduced *PHP Global Root Node* and *PHP Function Root Node* as call targets to execute our Truffle AST. PHP's language specification defines two different kinds of functions. We will now introduce *unconditionally defined functions* [12, Section 13-functions, General].

These functions are defined at the top level of a script. Furthermore, function calls can either be executed before or after their function definition. This implies that we may not yet know a function definition when reaching a function call. Our design proposes a dynamic lookup mechanism for function calls. In contrast to multiple AST traversals, a lazy lookup mechanism allows future work towards dynamic name binding[3]. The proposed class hierarchy is depicted in Figure 3.5.

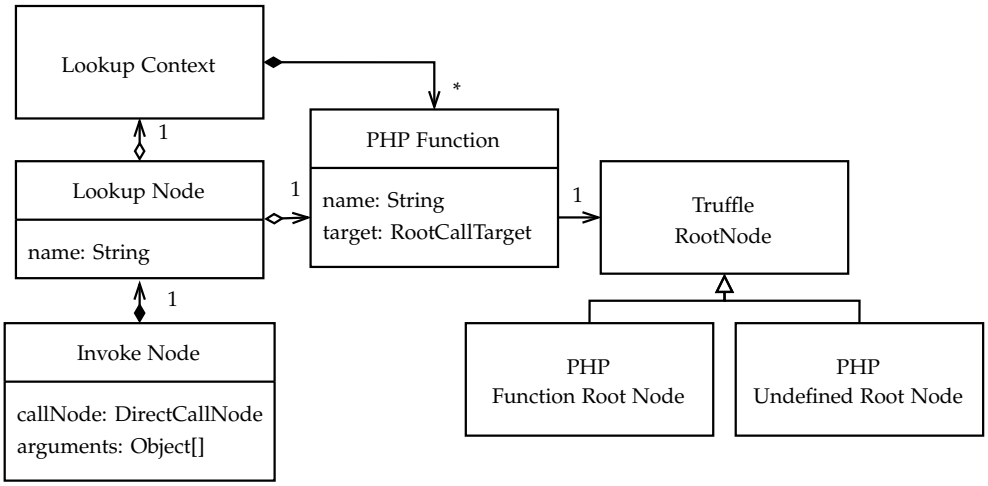

Figure 3.5: High level lazy function lookup mechanism.

When traversing the source AST, we create *Invoke Nodes* for all function calls. An *Invoke Node* contains a *Lookup Node* which resolves a function definition by name at runtime. As long as a function implementation is not available, i.e. we have not parsed its definition, we use a call target derived from a dummy implementation. Upon parsing the function body, we update the PHP function with the correct implementation. We will fail when the dummy implementation derived from *PHP Undefined Root Node* is invoked, indicating that an undefined function is called. We

[3]Dynamic name bindings conditionally define a function at runtime [12, Section 13-functions, General]

employ Graal's `@CompilationFinal`, `partialEvaluationConstant()` and `transferToInterpreterAndInvalidate()` primitives to lookup a function definition in the interpreter and then assume that their definition does not change. The lookup therefore only occurs once at a given call site and is removed when compiled to native code [32]. We implement the lookup with a map.

We can employ a Truffle *DirectCallNode* for calling the root node of a function. Graal can then inline *DirectCallNode*s into call graphs of the parent node and can emit optimized code [32]. There is no need for a polymorphic inline cache[4] when calling a function because PHP cannot redefine functions. Listing 5 shows an excerpt of the invoke implementation.

```
1  // Excerpt from Invoke Node
2  @Child PhpExprNode lookupNode;
3  @Children PhpExprNode[] arguments;
4  @Child @CompilationFinal DirectCallNode callNode;

5  @ExplodeLoop
6  @Override
7  public Object executeGeneric(VirtualFrame frame) {
8    if (this.callNode == null) { // lazy lookup
9      PhpFunction fun = lookupNode.executePhpFunction(frame);
10     CompilerDirectives.transferToInterpreterAndInvalidate();
11     this.callNode = DirectCallNode.create(fun.getCallTarget()); // (1)
12   }
13   // resolve arguments
14   CompilerAsserts.partialEvaluationConstant(arguments.length);
15   Object[] vals = new Object[arguments.length];
16   for (int i = 0; i < arguments.length; i++) {
17     vals[i] = arguments[i].executeGeneric(frame); // (2)
18   }
19   // call function
20   return this.callNode.call(vals);
21   }
22 }
```

Listing 5: Excerpt from Invoke Node. The DirectCallNode is assumed to be final. This removes the cost of the lookup once we lazily loaded the function implementation.

At (1) we transfer back to the interpreter because we change a field which we assumed to be final. This is the initial look-up cost. The use of `@ExplodeLoop`

[4]Polymorphic Inline Cache: Cache at the call site of a method invocation to speed up the lookup of the implementation.

and `partialEvaluationConstant` allow us to unroll the loop in (2) which removes the branch overhead of the loop. Function redefinition is not possible so we know that the number of invoke arguments is constant during execution.

3.3.3 Scope and Variables

The concept of *scope* in a programming language defines the range of visibility from where a variable can be accessed. The only scoping mechanisms available in PHP are *Function Scope* and *Global Scope* [12, Section 04-basic-concepts, Scope]. A variable defined within a function is only visible in that function, so called *Function Scope*, while variables defined at the top level of a script are in *Global Scope*.

The example in Listing 6 emphasizes on the concept of scoping. Variable `$a` is in *Global Scope* and not reachable within `foo()`. Additionally, there is no concept of scoping within blocks as other C-like languages may support[5]. The lack of *Block Scope* causes variable `$b` to be visible in the entire function from the moment it was first assigned.

```
1  <?php
2  $a = 1337;
3  foo();
4  function foo() {
5    echo $a;  // error, variable 'a' not defined
6    if(true) {
7      $b = 42;
8    }
9    echo $b; // valid, echo 42;
10 }
```

Listing 6: PHP's function scope.

We model the limited range of PHP's *Function Scope* with virtual frames provided by Truffle. A Truffle frame is conceptually similar to a stack frame and models an activation record of a function call. Frames store local variables and function arguments and receive type specializations. In the terminology of Truffle, we distinguish between *virtual-* and *materialized frames*. A virtual frame cannot be stored as a reference in a Java field and can be eliminated by Graal using escape analysis. Frames store variables internally in an array which is eliminated by connecting reads of a variable with their corresponding writes. Graal inlines `execute*(Frame)` methods under the same root and can emit efficient code for local variables [32]. The limited scope of PHP's *Function Scope* makes virtual frames a good fit.

[5] cf. block scope or lexical scope in other programming languages

3.3.4 Control Structures

We implemented control flow semantics for selections and iteration. This includes *if/else*, *do-while*, *for*, *while*, *break*, *return*, *continue*. We now portray one example, the *Ternary-Operator* and highlight key concepts in implementing control flow. Other features follow similar ideas.

Ternary Operator

The ternary operator is a short-hand construct for an if-else block. Instead of writing if-else, one may use the ternary expression shown below, where `$result` is assigned 1 if `$a` equals 1337.

```
1 <?php
2 $result = ($a == 1337) ? 1 : 2;
```

Listing 7: Ternary operator.

The use of the ternary operator in PHP can lead to confusion because most languages implement it with right-associativity. However, in PHP it is left-associative. We illustrate the issue in Listing 8. `$result` will contain the value 4 and not 2.

```
1 <?php
2 $result = 1 ? 2 : 3 ? 4 : 5;
3 // result: 4

4 $left  = (1 ? 2 : 3) ? 4 : 5;
5 // left: 4;

6 $right = 1 ? 2 : (3 ? 4 : 5);
7 // right: 2;
```

Listing 8: Nested ternary operator.

With PHP 7.4, the use of nested ternary operations without explicit brackets is deprecated and will be dropped in PHP 8. However, the operator is only rarely used under these circumstances [39]. Based on these reasons, our implementation bails out to enforce explicit brackets.

Due to dynamic typing, an expression in a condition can return a different type than boolean. Hence, we need to convert it to a boolean to select a branch. The specification states decision tables to convert values to true and false [12, Section 08-conversions, Converting to Boolean type]. In general, numeric values such as `0` and `0.0` are treated as false while other values are true. This implies that -1 for instance is considered as true, like any other non-zero number. For type conversions in conditions we propose *Conversion Node*s. In Figure 3.6, a *Convert2Boolean* acts as a proxy and will specialize

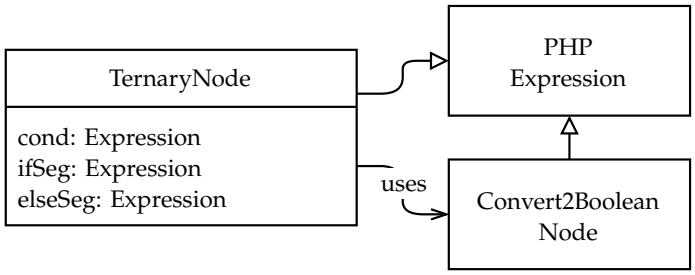

Figure 3.6: Overview ternary operator node.

on a type, *bool, long, double, PhpArray*, based on program behavior. We then convert values explicitly to boolean. Listing 9 portrays some of these type specializations.

```
1 @Specialization
2 boolean doBoolean(boolean val) { return val; }
3 @Specialization
4 boolean doLong(long val) { return val != 0; }
5 @Specialization
6 boolean doDouble(double val) { return val != 0.0; }
7 // ...
```

Listing 9: Excerpt from Convert2Boolean node.

For control flow optimizations, we employ Truffle's *ConditionProfile* and *BranchProfile* primitives to count the number of times conditions and branches are taken. Graal can then speculate on an outcome and exclude infrequently taken branches from JIT compilation.

3.3.5 Arrays

Arrays in PHP are universal data structures. They represent traditional integer indexed arrays, but can dynamically grow and generalize to support dictionary like features. Arrays can either contain integer or string indices, which may be negative or even discontinuous [12, Section 12-arrays, General]. Marr et al. [40] did a comprehensive analysis of collection designs in the wild. They identified six different collection types; *sequences*, *sets*, *maps*, *stacks*, *queues*, and *composed collections*. While this grouping is self-explanatory, with *composed collections* they refer to nested collections such as 2D arrays. According to this distinction, arrays in PHP are a facades for all six collection types. For instance, the use of runtime functions such as `array_shift` and `array_pop` allow arrays to be treated as *stacks and queues*.

Whereas a facade allows for universal use and hides implementation details,

it imposes challenges on an efficient implementation. Our array implementation exploits the common use case and allows for generalizations in case usage assumptions no longer hold. This follows the idea surrounding Truffle's specializations. In Figure 3.7 we illustrate our proposed design. To fulfill the semantic requirements imposed by our benchmarks, we implement a `long[]` and `Object[]` backend. Former implementation exploits Java's primitive long datatype and can store primitive integers without boxing, while the latter stores all other data types. This allows them to be *sequences*. Several other backends follow similar design ideas but are outside the scope of this work. The idea of self adapting storage representations is also contemplated by others. Bolz et al. for instance analyzed the implementation of different storage strategies for collections in the PyPy virtual machine [41].

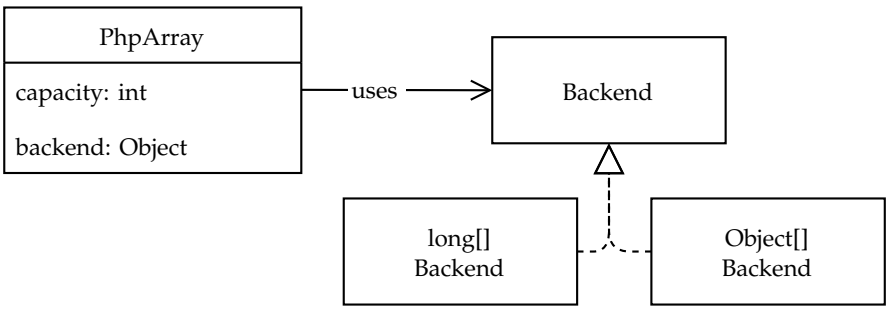

Figure 3.7: Concept of self adapting array backend implementations.

We introduce a new data type, `PhpArray`, which contains a capacity and a backend. The backend stores the array implementation which we change at runtime based on usage behavior. We model modifications on `PhpArrays` with *Truffle Libraries* and explicitly dispatch on the type of *PhpArray#backend*. This allows us to build a polymorphic dispatch mechanism with support for caching and profiling. With *@CachedLibrary*, we can cache a library and use it in Truffle node specializations. In terms of terminology, Truffle libraries are similar to Java interfaces where library messages correspond to interface methods. We can model a set of messages we want to support in our array implementation and then specialize on an implementation based on data types we observe at runtime [42]. We will walk through an example in the next section.

In Figure 3.8 we depict a simple use case. Our array initially assumes a `long[]` backend. While this suffices our implementation, we may add additional backends and introduce a *Placeholder backend* which does not assume a type. In Figure 3.8 we first create an array `$a = array()`, add an integer `$a[0] = 1`, and later on add a float. Upon insertion of the float value the assumptions no longer hold and we must generalize to an implementation which satisfies all values. This involves creating a new array and

copying array entries. Type generalizations only occur in a constant number of times. Additionally, we allocate backends with an initial size and dynamically grow them if storage capacity is exceeded. By doubling the array size upon reaching full capacity, array writes have a constant amortized time complexity. Read operations read from the under-laying array backend and are constant as well. Furthermore, the use of the PHP builtin function `array_fill` allows us to create arrays with a given initial size. In this case, arrays no longer dynamically grow to fulfill the requested capacity.

Having only two backends may lead to inefficient runtime behavior. For instance, at some point a map like data structure may be a better fit, in particular if arrays are only filled sparsely. As previously stated, more backends are outside the scope of this thesis.

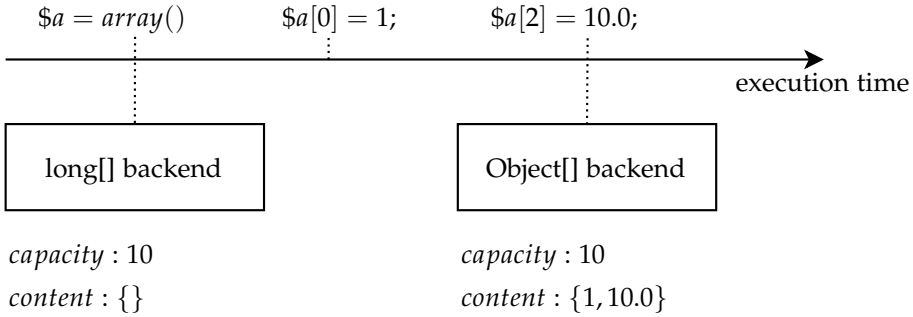

Figure 3.8: Example for array specialization based on usage.

3.3.6 Implementing Array Writes

Recall that arrays in PHP are dynamic and untyped. If we write outside the allocated array capacity we must extend the array. Furthermore, we must generalize to a new backend if an unsupported type is inserted.

In the next sections we will walk through the implementation of our arrays using Truffle libraries and aim to identify some of the implementation challenges. Subsequently, the AST node to write arrays is shown at the end in Listing 13.

When writing array entries, we distinguish between four cases.

1. Write an array entry in-bounds with the same type
2. Write an array entry in-bounds with type mismatch
3. Write an array entry out-of-bounds with the same type
4. Write an array entry out-of-bounds with type mismatch

Cases 2 and 4 require type generalizations because we write into a backend which cannot host the new value. We will portray case 4, however, other cases will follow similar ideas. To write out-of-bounds with type mismatch we will:

a. allocate a new backend with larger capacity,

b. copy all existing elements,

c. write the new array element, and finally,

d. expose array write functionality with an AST node.

a. Allocate a new Array Backend

We expose a new library message *GeneralizeForValue* which selects a compatible backend based on a given type. The message implementation for `long[]` is portrayed in Listing 10. We will either select a `long[]` allocator for a `long` (1) or a `Object[]` allocator for any other type (2).

```
1  @ExportMessage
2  static class GeneralizeForValue {

3    @Specialization
4    protected static Allocator
5      generalizeForValue(long[] receiver, long  newValue) { /* (1) */
6        return LongAllocator.INSTANCE;
7    }

8    @Specialization
9    protected static Allocator
10     generalizeForValue(long[] receiver,Object newValue) {  /* (2) */
11       return ObjectAllocator.INSTANCE;
12   }
13 }
```

Listing 10: Array library message *GeneralizeForValue*.

With *@ExportMessage* we instruct Truffle's annotation processor to generate a new implementation in the array library. An allocator is responsible to create new backends. We can test whether a given value can be stored in the array and create new backends with a given capacity. Figure 3.9 depicts the allocator design.

b. Copy existing Array Elements

Having an allocator in place, we now migrate array entries from the old backend. A library message *CopyContents* will write all entries into a new

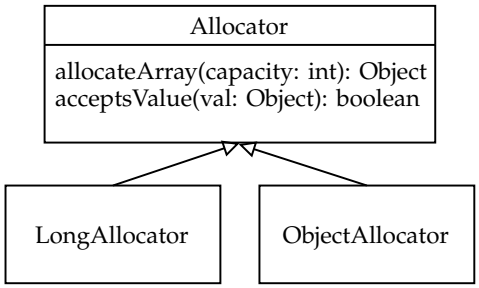

Figure 3.9: Allocator to create new PHP array backends.

backend. Note the use of Truffle's `@CachedLibrary` in Listing 11. We do not know the internal representation of the `destination` backend and delegate the write to the array library of `destination`. This decouples writing to arrays from their internal storage representation (1).

```
1  @ExportMessage
2  static class CopyContents {

3    @Specialization(limit = ArrayLibrary.SPECIALIZATION_LIMIT)
4    protected static void copyContents(
5      long[] receiver,
6      Object destination,
7      int length,
8      @CachedLibrary("destination") ArrayLibrary destinationLibrary) {
9        for (int i = 0; i < length; i++) {
10         destinationLibrary.write(destination, i, receiver[i]); // (1)
11       }
12   }
13 }
```

Listing 11: Library message *CopyContents* for `long[]`.

c. Insert new Array Element

Having copied all entries to a new backend, we can now finally write the new entry. The library message *Write* in Listing 12 implements the write on the `long[]` backend.

```
1  @ExportMessage
2  static class Write {
3    @Specialization
4    protected static void write(long[] store, int index, long value) {
5      store[index] = value;
6    }
```

```
}
```

Listing 12: Library message *Write* for `long[]`.

d. Expose Write Functionality with AST nodes

With the design in place to write to backends, we now model an AST node to write out-of-bounds with type-mismatch. Note the three steps (a), (b), (c) in Listing 13 which we previously introduced.

```
@NodeChild(value = "phpArray")
@NodeChild(value = "index")
@NodeChild(value = "value")
public abstract class ArrayWriteNode extends PhpExprNode {

  // Node to write array[index] = value
  public abstract Object executeWrite(
                  PhpArray array, long index, Object value);

  // write to array out-of-bounds and with type mismatch
  @Specialization(
      guards = {
          "!library.acceptsValue(array.getBackend(), value)"
          ,"isOutOfBounds(array, index)"
        },
    limit = LIMIT)
  protected Object writeOutOfBoundsTypeMismatch (
            PhpArray array,
            long index,
            Object value,
            @CachedLibrary("array.getBackend()") ArrayLibrary library,
            @CachedLibrary(limit = LIMIT) ArrayLibrary libraryNewBackend) {

    final int newCapacity = getNewCapacity(array, index); // doubles capacity
    final Object oldBackend = array.getBackend();

    // (a) Allocate a new backend
    final Object newBackend = library
                .generalizeForValue(array.getBackend(), value)
                .allocate(newCapacity);

    // (b) Copy existing Elements
    library.copyContents(oldBackend, newBackend, array.getCapacity());
    array.setBackend(newBackend);
    array.setCapacity(newCapacity);

    // (c) Insert new Element
    libraryNewBackend.write(array.getBackend(), convertToInt(index), value);
    return value;
```

```
34    }

35    protected static boolean isOutOfBounds(PhpArray array, long index) {
36      return array.getCapacity() <= index;
37    }
38  }
```

Listing 13: Excerpt from array write node, write-out-of-bounds with type mismatch.

3.3.7 Pass-by-Reference

Arrays are by default copied by-value and target to runtime optimization to avoid unnecessary memory copies [12, Section 04-basic-concepts, Deferred Array Copying]. In one of our synthetic benchmarks we allocate deeply nested array entries (Section 4.1.2). In order to demonstrate the performance of our implementation, we provide two source versions of these benchmarks. One version is the original and the other version uses explicit pass-by-reference behavior.

PHP supports explicit pass-by-reference semantics with the reference operator (*&*). The operator is defined at the call site, return type and parameter declaration. In Listing 14, line 9 enforces an array copy of `$tree`, while line 7 and 8 do not[6].

```
1  <?php
2  function &createTree($depth) {
3    $a = array();
4    // ...
5    return $a;
6  }
7  $tree = &createTree(1337); // copy-by-reference
8  $sameTree = &$tree;        // copy-by-reference
9  $copyTree = $tree;         // copy-by-value
```

Listing 14: Copy-by-reference at call site and return value.

We implement the reference operator for arrays. An approach for other data types is discussed in Future Work. Our proposed design is based on a proxy node which either employs by-value or by-reference behavior. Depending on the declared semantics, the proxy will enforce a deep copy or simply forward the reference of the array. For creating a deep copy we will allocate a new array backend and recursively copy all children. This is needed to correctly account for nested arrays. Our implementation proposes a new

[6]Supposing no optimizations are employed to copy-on-write or other techniques

library message (see Section 3.3.6 for an introduction to libraries) which recursively calls itself if an array entry contains another array.

3.4 Remaining Language Features

In the last sections we presented highlights in designing PHP on GraalVM. As we have already established, the main aim of our work is to perform a preliminary performance comparison of a GraalVM-hosted implementation. The selected features were therefore imposed by our benchmarks. PHP is an expressive programming language with a variety of different features. While we refer to Chapter 6 for general thoughts on future work, we will now give an overview of important remaining features. For a comprehensive overview we refer to the specification [12]. The snippet in Listing 15 demonstrates some of the implemented language concepts.

Datatypes In our experimental runtime we implemented boolean, integer, float and array data types. PHP supports ten datatypes including string, resource and callable [38]. The resource datatype, for instance, models a reference to an external resources such as a file handle and database connections. Other datatypes will follow similar ideas as we have established in this chapter. Challenges will include how to interoperate our data types with implementations of third-party code. We will talk about third-party libraries in Chapter 6.

Variables We implemented a subset of variables supported by PHP. Our implementation currently supports local variables. While these are the most used variables, PHP further supports static, and (super) global variables. We intend to implement these with maternalized Truffle frames, analogously to our existing local variable implementation.

Opt-In Type system With PHP 7, an optional stricter type system was introduced. Our parser correctly recognizes type declaration but does not yet propagate type declarations to our runtime. In further work we intend to implement this feature. On type mismatch, we can bail out instead of following conversion rules.

Classes A major language feature is support for object oriented programming. Wöss et al. introduced an object storage model for Truffle [43]. The model provides language agnostic tools to implement class hierarchies and object instantiation. PHP supports single inheritance and traits. In future work, we intend to evaluate how we can model object-oriented programming in Truffle.

Arrays Our array implementation demonstrates the potential performance gains. We implemented sequences. In future work we must implement map, stack and queue like structures and implement mechanisms to

copy-on-write and pass-by-reference automatically if semantically possible. These concepts will follow the same ideas with Truffle Libraries which we introduced in this work.

Reference Operator A major implementation aspect is the reference operator for primitive data types. Although HHVM[7] dropped support for references on primitive data types [44], it is still a major feature of the PHP programming language. We intend to investigate how this could be efficiently implemented. One approach involves boxing referenced primitive data types into Java objects. For instance, one design suggestion involves modeling referenced integer types in an Integer class which can be shared and modified by AST nodes.

Namespaces, Exceptions, Generators, Anonymous Classes We believe it is a matter of investing more engineering efforts to implement these features. Namespaces and parsing of multiple files will be an important feature towards compatibility with existing PHP applications.

```
1  <?php
2  function Fannkuch($n){
3    $p = $q = $s = array();
4    $sign = 1; $maxflips = $sum = 0; $m = $n-1;
5    for ($i=0; $i<$n; $i++){ $p[$i] = $i; $q[$i] = $i; $s[$i] = $i; }
6    do {
7    // Copy and flip.
8      $q0 = $p[0];                                  // Cache 0th element.
9      if ($q0 != 0){
10       for($i=1; $i<$n; $i++) $q[$i] = $p[$i];    // Work on a copy.
11         $flips = 1;
12           do {
13             $qq = $q[$q0];
14             if ($qq == 0){                 // ... until 0th element is 0.
15               $sum += $sign*$flips;
16               if ($flips > $maxflips) $maxflips = $flips;  // New maximum?
17                 break;
18               }
19               $q[$q0] = $q0;
20               if ($q0 >= 3){
21                 $i = 1; $j = $q0 - 1;
22                 do {
23                   $t = $q[$i]; $q[$i] = $q[$j]; $q[$j] = $t; $i++; $j--; }
24                 while ($i < $j);
25               }
26             $q0 = $qq; $flips++;
27           } while (true);
28         }
29         // Permute.
```

[7] Virtual machine to execute Hack, a dialect of PHP

```
30          if ($sign == 1){
31            // Rotate 0<-1.
32            $t = $p[1]; $p[1] = $p[0]; $p[0] = $t; $sign = -1;
33          } else {
34            // Rotate 0<-1 and 0<-1<-2.
35            $t = $p[1]; $p[1] = $p[2]; $p[2] = $t; $sign = 1;
36            for($i=2; $i<$n; $i++){
37              $sx = $s[$i];
38              if ($sx != 0){ $s[$i] = $sx-1; break; }
39              if ($i == $m){
40                return array($sum,$maxflips); // Out of permutations.
41              }
42              $s[$i] = $i;
43              // Rotate 0<-...<-i+1.
44              $t = $p[0];
45              for($j=0; $j<=$i; $j++){ $p[$j] = $p[$j+1]; } $p[$i+1] = $t;
46            }
47          }
48    } while (true);
49  }
```

Listing 15: Excerpt from a function in benchmark *fannkuchredux* to demonstrate some of the implemented features [45].

Chapter 4

Experimental Methodology & Evaluation

In the following chapter we compare performance of *graalphp* with other implementations of PHP. We first introduce terminologies and performance metrices. We then proceed to outline benchmarks before we present and discuss the measurement results. These experiments aim to perform a preliminary performance comparison, given that the implementation is not yet fully realized. Hereby, our goal is to investigate potential speedups.

4.1 Experimental Methodology

4.1.1 Performance Metrices

A program is said to reach *peak performance* once it is in *steady-state*. *Steady-state* is a state during program execution after the dynamic compiler gathered enough profiling information to translate frequently executed code segments to machine code. Frequently executed code fragments are also called *hot*. Time spent before reaching *steady-state* is often referred to as *warm-up time* [46]. The metric we use to evaluate performance is *peak performance*.

Benchmarking Java applications is often challenging because a number of non-deterministic factors may influence performance. Some of these factors include dynamic compilation, garbage collection, thread scheduling and VM optimizations. As Georges et al. indicate, being rigorous in the experimental methodology is crucial because different methodologies may lead to different conclusions [47]. Hence, we take care in adding rigor. Apart from visualizing data in plots we will also present timing results to support readers in drawing their own conclusions.

Collecting Samples We rely on language runtime APIs to obtain timing values. For PHP this involves calling `hrtime(true)`. A code snippet for a

single iteration is shown in Listing 16. Georges et al. recommend running multiple consecutive benchmark iterations within a single VM execution, as well as running multiple VM executions [47]. We will follow their method and list total number of iterations and number of VM executions when presenting timing results. Kalibera et al. establish different points of repetition as *levels* of the experiment [48]. According to their distinction, the lowest level is a single iteration of a benchmark, while the highest level is compilation of VMs and other binaries. In order to keep execution within a reasonable time margin, we will measure at a *low level* and will not recompile virtual machines and other runtime binaries across benchmark executions.

```
1  <?php
2  for($i = 0; $i < $N; $i ++) {
3      $start = hrtime(true);
4      $val = doIteration();
5      $timing = hrtime(true) - $start;
6    }
```

Listing 16: Sampling N timing values in PHP

Aggregating Samples We aggregate samples obtained from the benchmarks using arithmetic mean. For better interpretability of the results we further enumerate minimum, maximum, sample standard deviation and confidence interval for the mean.

Determining when *steady-state* is reached during program execution is often challenging [46]. Dynamic compilation is typically performed within the first iterations which is why we manually inspect iterations to pick a threshold and exclude *warm-up* samples before aggregating.

The sample mean $\bar{T}$ and sample standard deviation s_n are used as follows,

$$\bar{T} = \frac{1}{n} \times \sum_{i=1}^{n} t_i, \tag{4.1}$$

$$s_n = \sqrt{\frac{1}{n-1} \times \sum_{i=1}^{n} (t_i - \bar{T})^2}, \tag{4.2}$$

for n samples t_1 to t_n.

For the confidence interval we chose the significance level $\alpha = 0.95$ and determine c_1 and c_2 such that

$$Pr[c_1 \leq \mu \leq c_2] = 1 - \alpha, \tag{4.3}$$

$$c_1 = \bar{T} - z_{1-\alpha/2} \frac{s_n}{\sqrt{n}}, \tag{4.4}$$

$$c_2 = \bar{T} + z_{1-\alpha/2} \frac{s_n}{\sqrt{n}}, \tag{4.5}$$

where μ is the actual mean of the population, and $z_{1-\alpha/2}$ defined such that a random variable $\mathbb{Z}$ which is Gaussian distributed ($\mu = 0$, $\sigma^2 = 1$) follows

$$Pr[\mathbb{Z} \leq z_{1-\alpha/2}] = 1 - \alpha/2. \tag{4.6}$$

$z_{1-\alpha/2}$ is obtained from a precomputed table [47].

Put differently, with a significance level of $\alpha = 0.95$ we know that every time we repeat the experiment and obtain a new sample mean $\bar{T}$, in 95% of these cases the real mean[1] of the benchmark lies within the endpoints c_1 and c_2 calculated from the samples.

The underlying assumptions are that the samples are independent and come from the same population. In practice, the samples may not be independent because they may modify heap state, cache entries and other data structures [47]. To account for this, we remove *warm-up* samples from the measurements.

Speedup Large differences in execution time become harder to visualize when absolute timing measurements are used. We therefore compare aggregated samples relative to a baseline. The baseline used is the current version of PHP which we further specify in Section 4.1.3. Implementation A is s times faster than PHP with speedup s defined as

$$s = \frac{\bar{T}_A}{\bar{T}_{PHP}}, \tag{4.7}$$

$\bar{T}_X$ is the mean for implementation X. The relative performance gain is $s - 1$, or put as an example, a speedup of 1.5 leads to 50% performance improvements.

Warm-up In addition to speedup of peak performance, we will include box plots to show warm-up time. These plots will indicate how many iterations are needed until we reach peak performance.

4.1.2 Synthetic Benchmarks

The implemented features of graalphp were dictated by language features used in *The Computer Language Benchmarks Game* [11]. Namely we implemented three of the benchmarks available in PHP: *binary-trees*, *spectralnorm*

[1] cf. population mean in statistics

and *fannkuchredux* (presented in Table 4.1). A list of syntactic modifications can be found in Appendix C.1. Modifications were necessary to keep the feature set within the scope of this thesis. We believe that our modifications do not significantly influence the outcome of the experiments because they only remove syntactic sugar. To state an example, instead of the *for each* operator we use the *for* operator to iterate over an array. Additionally, our selected benchmarks neither involve file I/O nor do they increase performance with threads. The parameter *N* given in Table 4.1 is dictated by the Computer Language Benchmarks Game.

Benchmark	Language Features	Parameters
binary-trees [49]	functions, nested arrays, pass-by-reference/value, integers, unset, loops	N = 21
spectralnorm [50]	functions, arrays, loops, floats	N = 5500
fannkuchredux [45]	functions, arrays, loops, integers	N = 12

Table 4.1: Language features and configurations used in benchmarks.

Pass-By-Reference and Pass-By-Value

When presenting results we will provide two source versions of the benchmarks; one version implements by-value semantics for arrays and the other version implements by-reference semantics. By-reference explicitly uses the *Reference Operator* of PHP (& Operator). Arrays are by default passed by-value and are target to runtime optimization to avoid unnecessary memory copies [12, Section 04-basic-concepts, Deferred Array Copying]. We will provide two source versions to allow *graalphp* to explicitly pass arrays by-reference and hence avoids deeply nested array copies. A more sophisticated array implementation may employ *Copy-On-Write*, *Points-to Analysis* and similar techniques to automatically distinguish between these two. We provide two source versions to show what can be achieved if more engineering efforts are put in our implementation.

4.1.3 Runtime Implementations

All benchmarks are executed in a Linux container image based on Ubuntu 20.04 (Docker/ Podman). This is motivated to simplify reproducibility and installation of development and benchmark dependencies. We did not notice an overhead when benchmarking in the container environment.

graalphp/graaphp-native Graalphp runs on GraalVM Community Edition 20.1.0. Additionally we provide graalphp-native, an AOT compilation of graalphp using GraalVM Native Image [28]. We include a native build to indicate some of the potential of AOT compilation. However,

no efforts were spent to further optimize the native build. The binaries are built based on git commit *cb59d053633d*.

PHP 7.4 We run all benchmarks against the current version of PHP. At the time of writing this report, this was PHP 7.4.3. Results of PHP 7.4 with copy-by-value semantics are used as a baseline for speedup diagrams.

PHP 8.0 Alpha A JIT compiler for PHP is scheduled for PHP version 8 which is announced to be available by the end of 2020[2]. We compiled the latest tagged alpha version (php-8.0.0alpha3) from Github[3] and enabled the JIT compiler with flags `opcache.jit=1235`, `opcache.jit_buffer_size=512M`.

Hack/HHVM The HipHop Virtual Machine (HHVM) is the runtime environment for the Hack programming language, a dialect of PHP. HHVM is developed by Facebook and utilizes a JIT compiler. Some of Hack's language feature include dynamic and static typing as well as generics. HHVM is heavily used by Facebook internally and is open source software [3]. We run version 4.68.0 (rel) and modified our benchmarks accordingly. Only small changes were needed which include adding type annotations and replacing PHP arrays by Hack vectors.

JPHP JPHP is an alternative implementation of PHP for the JVM. The project dynamically transforms PHP source code to Java bytecode which is then executed on the JVM. For bytecode generation the ObjectWeb ASM[4] framework is used. JPHP implements a subset of the PHP standard libraries and provides custom extensions [51]. We use the latest released version: jppm 0.6.7 which bundles jphp 1.0.3, OpenJDK 14.

4.1.4 Hardware Setup

We execute benchmarks on a ThinkPad P52 running Manjaro Linux Minimal (Architect) Installation with Linux Kernel 5.7. The hardware features an Intel® Xeon® E-2176M CPU clocked at 2.7GHz - 4.4GHz and 16GB DDR4 RAM. The device has Hyper-Threading disabled, Turbo Boost disabled, and CPU frequency driver *intel_pstate* is used with governor *performance* manually set at a fixed clock rate of 2.7 GHz. We further restart the device before running a new benchmark suite[5].

[2]PHP 8 Release Plan `https://wiki.php.net/todo/php80`, *archived: url*
[3]PHP 8: `https://github.com/php/php-src/tree/php-8.0.0alpha3`, *archived: url*
[4]Java bytecode manipulation framework: `https://asm.ow2.io/`, *archived: url*
[5]By benchmark suite we mean the set of all benchmarks

4.2 Evaluation

In this section we present our preliminary results on peak performance of the selected benchmarks. Figures 4.1, 4.2 and 4.3 show execution times normalized to PHP 7 and Figures 4.4 and 4.5 depict evaluation of warm-up time for the first iterations. Subsequently, we list absolute timing measurements in Tables 4.2, 4.3 and 4.4 at the end of the chapter.

We see competitive to significantly faster speedup results of *graalphp* across all selected benchmarks. For *Fannkuchredux* (Figure 4.1) we notice that *graalphp* is much faster than all benchmarked implementations. We reach a speedup of 859% compared to PHP 7. The AOT build of our implementation – *graalphp-native* – reaches a speedup of 705%. Note that no arrays were used in this benchmark which is why we only present the benchmark without the reference operator.

Both *spectralnorm* (Figure 4.2) and *binary-trees* (Figure 4.3) use semantics for copy-by-value (blue, default) and by-reference (green). In those, *graalphp* exceeds PHP 7 in *spectralnorm* by 652% (copy-by-value) and 655% (copy-by-reference). For *binary-trees* we are worse with deeply nested array copies by -84% (copy-by-value) and exceed all implementations except HHVM in copy-by-reference with a speedup of 511%.

Copy-by-Reference semantics demonstrates the upper bound in performance we can achieve if more engineering efforts are spent in an array implementation. Techniques such as *Copy-On-Write*, *Points-to*, and *Escape Analysis* will help to prevent unneeded array copies. This is however, outside of the scope of our work. Additionally, we note that other implemen-

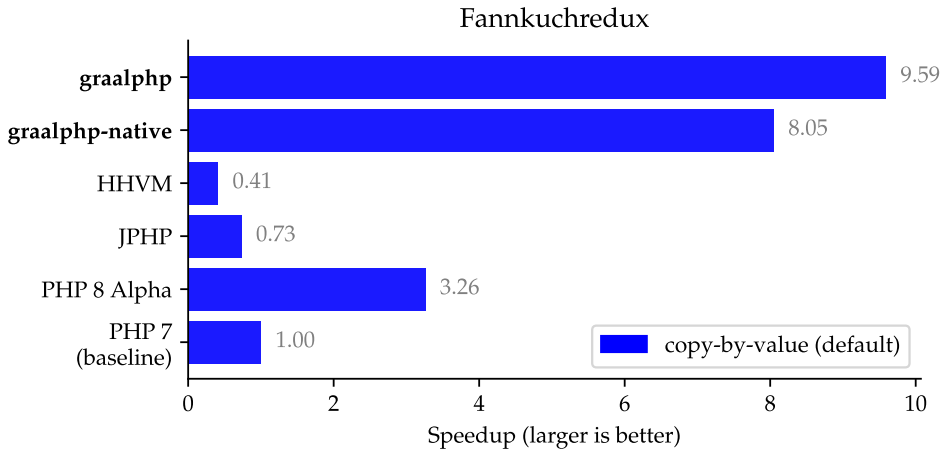

Figure 4.1: Speedup *fannkuchredux*, the x-axis shows speedup relative to PHP 7, while the y-axis show the implementation of PHP. Larger is better.

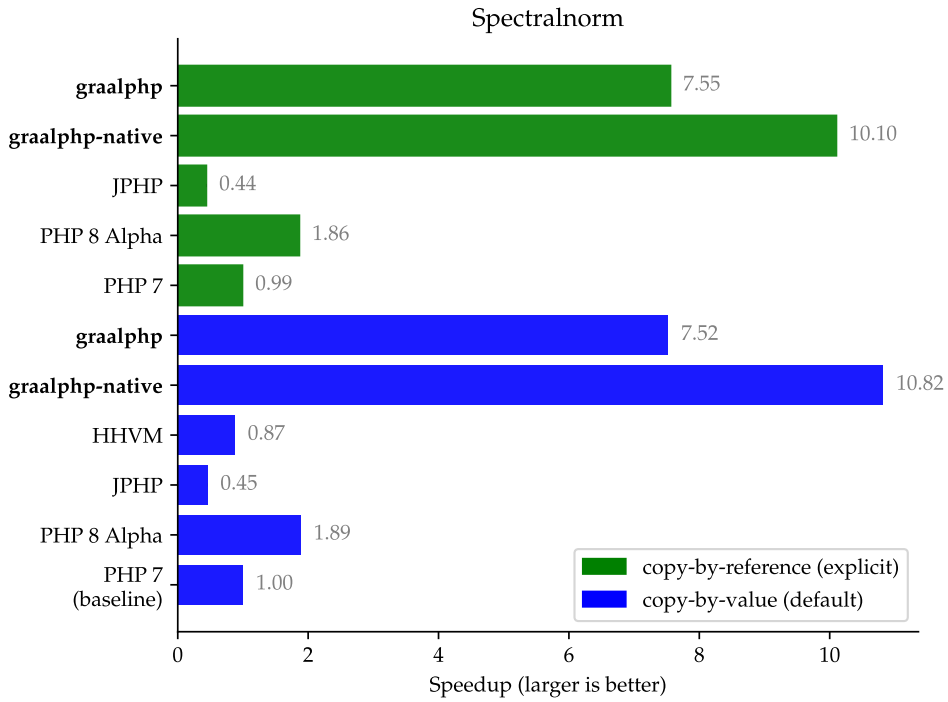

Figure 4.2: Speedup *spectralnorm* relative to PHP 7. Larger is better.

tations apart from *graalphp* and *graalphp-native* do not show much difference between by-value and by-reference behavior. Zend Engine (PHP 7, 8) for instance handles variables internally with a *Copy-On-Write* implementation which is why stating by-reference semantics may restrict compiler optimizations. For newer versions of Zend Engine it is often discouraged to pass values by-reference for performance reasons only [52]. HHVM discontinued support for by-reference semantics in 2019[6] when dropping source compatibility for PHP. This leaves more optimization opportunities completely up to their compiler.

[6] `https://hhvm.com/blog/2019/10/01/deprecating-references.html`, *archived: url*

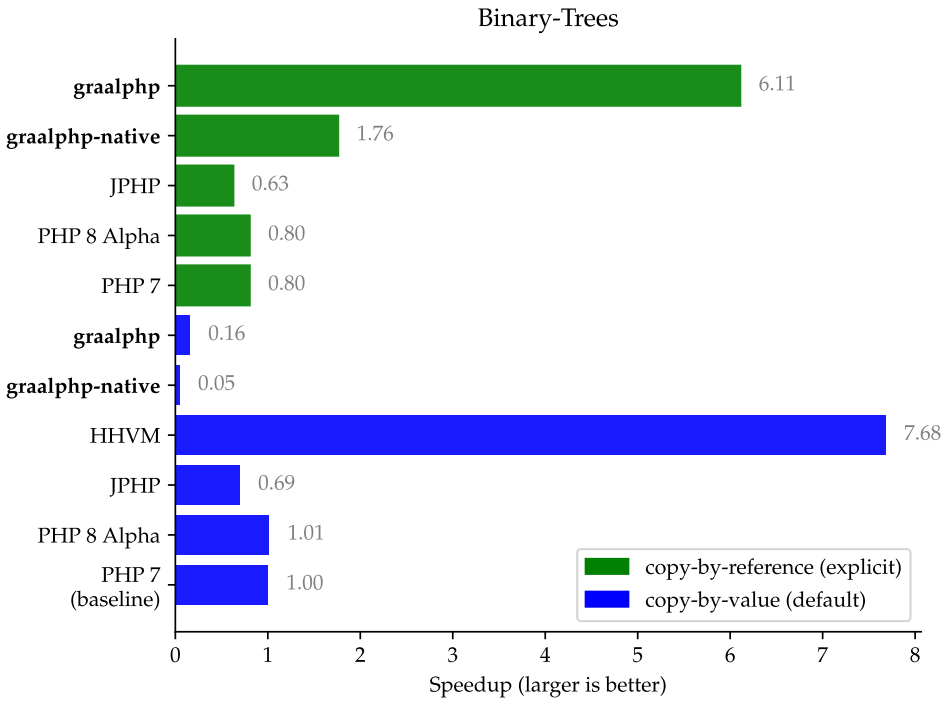

Figure 4.3: Speedup *binary-trees* relative to PHP 7. Larger is better.

Figure 4.4 and 4.5 show evaluation of warm-up time of *graalphp* and *graalphp-native* on the selected benchmarks. We see that execution time stabilizes within the first four iterations without much variance in *fannkuchredux* and *spectralnorm*. For *binary-trees* we see more variance. For the native build – *graalphp-native* – we notice that execution time increases after the first iterations in *binary-trees*. This benchmark stress tests garbage collection by allocating deeply nested arrays. The increase does not happen in *graalphp* which leads us to question whether it is due to lack of optimizations on native build settings, or different constraints imposed by SubstrateVM[7], which hosts the ahead-of-time compiled Java code. As we stated previously, optimizations on the native build are outside of the scope of this work.

Synthetic benchmarks are often employed as optimization targets for compilers [46], as well as other GraalVM/Truffle related research (e.g [53, 54, 55, 5]). On one hand, they are well studied in various programming languages. On the other hand, micro-benchmarks may not represent typical real-world program behavior. On the contrary, hardware and compilers may specifically be optimized to execute synthetic benchmarks and their performance may be worse in more realistic scenarios [46, 56]. For PHP, typical

[7]VM for ahead-of-time compiled Java code built with GraalVM Native image [28]

real-world program behavior includes server-side page generation which is subject to future work. Nonetheless, our results provide promising initial in-sighs into a GraalVM-hosted implementation of PHP. We significantly exceed all implementations in *fannkuchredux* and *spectralnorm*. For *binary-trees*, we show competitive results provided we can employ by-reference semantics and avoid deeply nested array copies. If we copy all arrays by-value we perform worse than PHP 7 in that benchmark. For comprehensive evaluation results, additional engineering efforts must be spent in supporting more language and runtime features. We will discuss this further in Chapter 6.

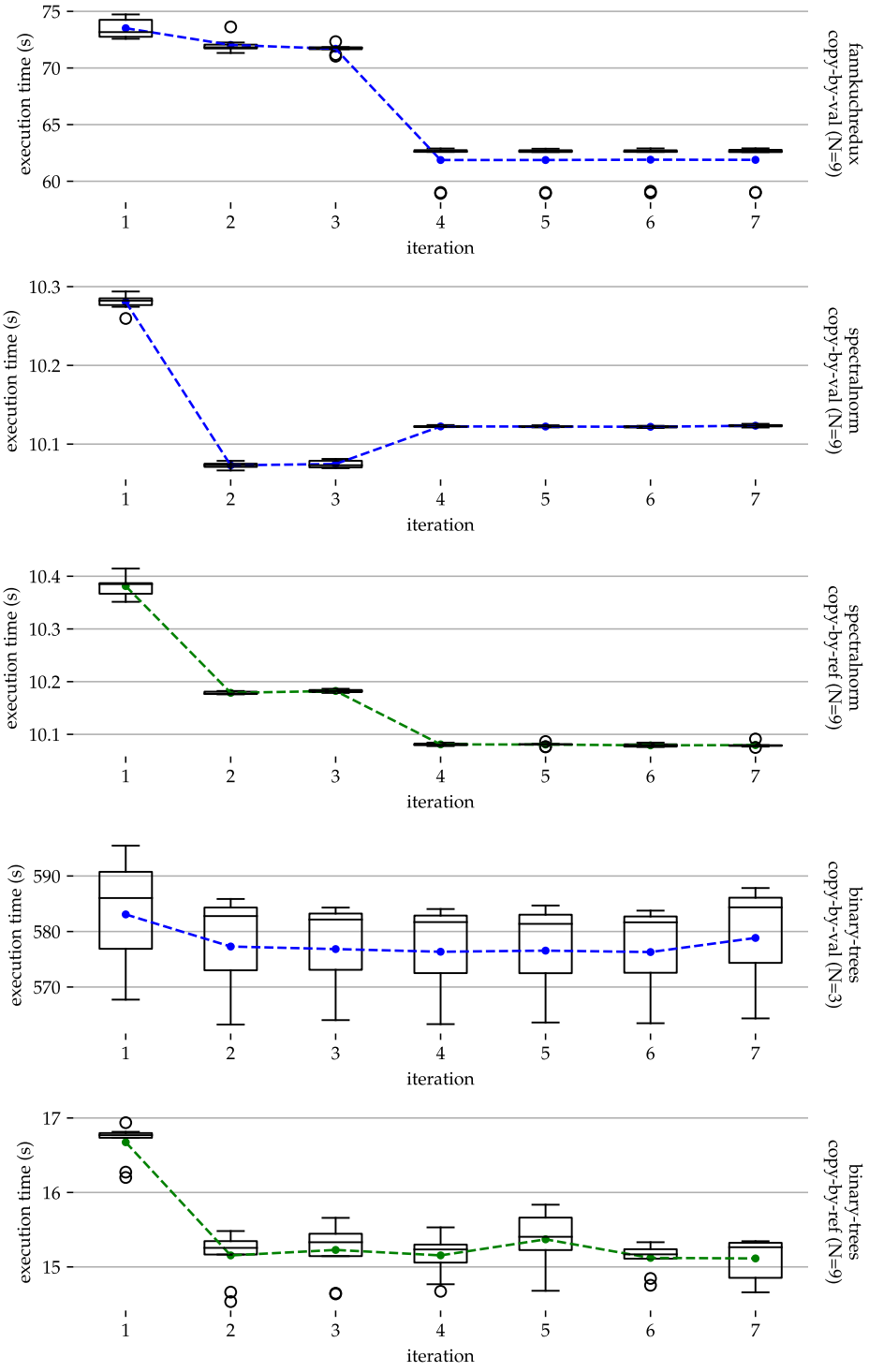

Figure 4.4: Evaluation warm-up time of *graalphp*. The x-axis shows the iteration of consecutive benchmark executions, while the y-axis shows the execution time of the benchmark. The *N* in the legend of the right y-axis refers to the number of VM invocations to run the consecutive benchmarks. Blue are copy-by-value, green are copy-by-reference versions of the benchmarks. Lower is better.

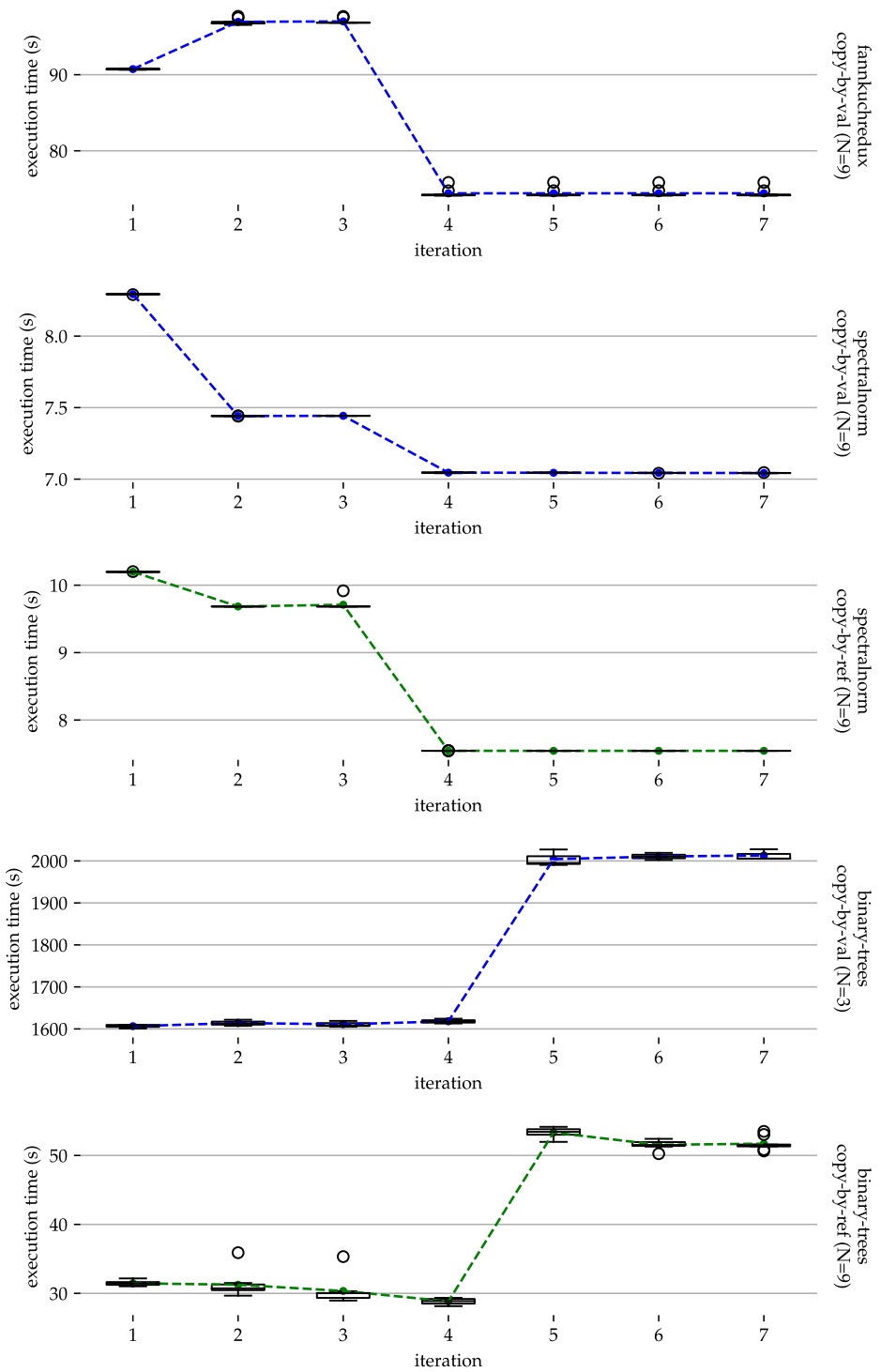

Figure 4.5: Evaluation warm-up time of *graalphp-native*. The x-axis shows the iteration of consecutive benchmark executions, while the y-axis shows the execution time of the benchmark. The *N* in the legend of the right y-axis refers to the number of VM invocations to run the consecutive benchmarks. Blue are copy-by-value, green are copy-by-reference versions of the benchmarks. Lower is better.

Fannkuchredux Peak Performance

Implementation	#	Min(s)	Mean(s)	Max(s)	Stdev	CI (0.95)	E
graalphp	90	58.92	62.34	62.91	1.15	[62.10; 62.58]	10
graalphp-native	90	74.11	74.29	75.86	0.40	[74.20; 74.37]	10
PHP 8 Alpha	20	175.64	183.42	195.22	7.38	[180.19; 186.65]	2
PHP 7 (baseline)	38	596.44	597.96	608.34	3.00	[597.00; 598.91]	4
JPHP	20	815.06	816.17	820.24	1.63	[815.46; 816.89]	2
HHVM	20	1455.64	1456.50	1457.48	0.88	[1456.11; 1456.88]	2

Table 4.2: Measurements for *Fannkuchredux* benchmark, excluded warm-up iterations: 5, sorted by mean. # refers to the total number of samples, while *E* refers to the number of VM executions to obtain these samples. Measurements presented in this table are visualized in Figure 4.1. *E* is typically higher for *graalphp* and *graalphp-native* because we also added peak performance samples obtained from benchmarking warm-up time (cf. Figure 4.4, 4.4).

Spectralnorm Peak Performance

Implementation	Type	#	Min(s)	Mean(s)	Max(s)	Stdev	CI (0.95)	E
graalphp-native		140	7.03	7.04	7.05	< 0.01	[$\sim$ 7.04; $\sim$ 7.04]	10
graalphp-native	by-ref	140	7.54	7.54	7.55	< 0.01	[$\sim$ 7.54; $\sim$ 7.54]	10
graalphp	by-ref	140	10.07	10.08	10.10	< 0.01	[$\sim$ 10.08; $\sim$ 10.08]	10
graalphp		140	10.12	10.12	10.13	< 0.01	[$\sim$ 10.12; $\sim$ 10.12]	10
PHP 8 Alpha		50	40.33	40.38	40.41	0.02	[40.37; 40.38]	2
PHP 8 Alpha	by-ref	50	40.84	40.87	40.91	0.02	[40.87; 40.88]	2
PHP 7 (baseline)		75	75.93	76.15	76.58	0.17	[76.11; 76.19]	3
PHP 7	by-ref	75	76.68	76.97	79.26	0.32	[76.89; 77.04]	3
HHVM		50	86.88	87.10	87.33	0.22	[87.04; 87.17]	2
JPHP		50	164.93	167.46	171.98	1.59	[167.02; 167.90]	2
JPHP	by-ref	50	171.55	174.64	181.27	2.59	[173.92; 175.36]	2

Table 4.3: Measurements for *Spectralnorm* benchmark, excluded warm-up iterations: 5, sorted by mean. *by-ref* use explicit pass by reference semantics. # refers to the total number of samples, while *E* refers to the number of VM executions to obtain these samples. Measurements presented in this table are visualized in Figure 4.2.

Binary-Trees Peak Performance

Implementation	Type	#	Min(s)	Mean(s)	Max(s)	Stdev	CI (0.95)	E
HHVM		90	11.66	11.83	12.29	0.12	[11.80; 11.85]	2
graalphp	by-ref	87	14.52	14.87	15.50	0.30	[14.81; 14.94]	15
graalphp-native	by-ref	84	50.26	51.61	54.21	0.77	[51.44; 51.77]	14
PHP 8 Alpha		90	84.55	89.64	92.22	1.98	[89.23; 90.05]	2
PHP 7 (baseline)		117	85.32	90.85	97.27	2.28	[90.44; 91.27]	3
PHP 8 Alpha	by-ref	90	107.86	113.00	118.61	2.05	[112.58; 113.42]	2
PHP 7	by-ref	90	106.99	113.01	117.51	2.24	[112.55; 113.48]	2
JPHP		90	124.30	130.75	138.81	3.38	[130.05; 131.45]	2
JPHP	by-ref	90	140.11	144.86	156.94	4.67	[143.89; 145.82]	2
graalphp		34	563.48	583.16	599.99	8.59	[580.27; 586.04]	6
graalphp-native		25	1999.98	2011.12	2029.56	7.69	[2008.11; 2014.14]	5

Table 4.4: Measurements for *Binary-Trees* benchmark, excluded warm-up iterations: 5, sorted by mean. *by-ref* use explicit pass by reference semantics. # refers to the total number of samples, while *E* refers to the number of VM executions to obtain these samples. Measurements presented in this table are visualized in Figure 4.3.

Chapter 5

Related Work

There has been numerous work by researchers and practitioners to examine and improve the performance of PHP. In this chapter, we first introduce *Zend Engine*, PHP's reference implementation and then distinguish between existing approaches to Source-to-Source translate and to Just-in-Time compile PHP programs. Subsequently, we identify other work on Truffle and GraalVM.

5.1 Zend Engine

The *Zend Engine* is PHP's reference implementation. It is a virtual machine written in C and runs on PHP opcodes, an intermediate representation (IR) similar to Java bytecode. *Zend Engine* is modeled as a bytecode interpreter. Upon parsing a script, the VM translates PHP code into its internal representation. To speed up this process, extensions such as *APC*[1] exist which act as a cache for parsing. *Zend Engine* stores function and class method definitions in *zend_op_array*[2]. This struct contains a pointer to the bytecode instructions *zend_op*[3] and various auxiliary data such as lookup tables for variables. *zend_op* represents a three-operand bytecode instruction with two arguments and a result operand. There are ca. 200 different bytecode instructions[4]. Zhao et al. [57] identified that *Zend Engine* may suffer performance overhead due to the mostly untyped nature of its bytecode. For dynamic typing, *Zend Engine* keeps values in generic boxed operands (*zend_op*), and performs type checking for access. A technical write-up on internal improvements on

[1] opcode cache: `https://www.php.net/manual/en/book.apc.php`, *archived: url*

[2] zend_op_array: `https://github.com/php/php-src/blob/master/Zend/zend_compile.h#L413`, *archived: url*

[3] zend_op: `https://github.com/php/php-src/blob/master/Zend/zend_compile.h#L142`, *archived: url*

[4] list of opcodes: `https://github.com/php/php-src/blob/master/Zend/zend_vm_opcodes.h`, *archived: url*

PHP 7 and its internal representation is curated in [58, 59, 60]. There is currently no JIT compiler in *Zend Engine* for PHP 7+. However, a JIT is planned for PHP 8. We use *Zend Engine* as baseline for evaluating *graalphp* because it is the reference implementation of PHP.

5.2 Source-to-Source Compilers

A popular approach to implement a compiler is to rely on mature existing compiler technology by translating the program to the source language of the existing compiler. This approach saves engineering efforts and is favorable if language concepts can easily be modeled in the target language [19]. In literature, source-to-source translation is also known as transpiling or transcompiling [61]. Transcompilers exist for a variety of programming languages including *cfront* [62], the original compiler for C++ (C++ to C), as well as *TypeScript* [63], a statically typed superset of JavaScript (TypeScript to JavaScript). For PHP, work on transpiling includes *HPHPc* [2] (PHP to C++), *phc* [1] (PHP to C), and *Roadsend PHP* [64] (PHP to Bigloo Scheme). These projects are transpiled to a statically typed target language and benefit from ahead-of-time optimizations of their respective compilers but are challenged to model dynamic runtime behavior such as *eval* or dynamic function definitions [65]. This issue is avoided by dropping support for certain language features or to implement an interpreter and transfer execution to it for these cases [65, 66]. The last release of *phc* dates back to 2009[5], work on *HPHPc* was discontinued in favor of *HHVM* (see next section), and according to their project site, development on *Roadsend PHP* has stopped[6].

As discussed by Romer et al. [19], issues surrounding ahead-of-time compilations of dynamic languages are prevented with interpreters and just-in-time compilation. We will introduce work on JIT compilers for PHP in the next section. Our approach partially follows methods of *Source-to-Source* compilers, by translating PHP code to a Truffle AST and then relying on mature infrastructure of Truffle and Graal. However, in contrast to the presented *Source-to-Source* approaches, we can utilize an interpreter and a JIT compiler to execute the program and thus, avoid issues surrounding direct AOT compilation.

5.3 Just-In-Time Compilers

Related work on JIT compilers for PHP includes *P9* [67] which adapted an existing JIT to run PHP. They used *TR JIT*, a dynamic compiler from an IBM JVM implementation. *Quercus* [68] and *JPHP* [51] follow a similar approach

[5] *phc*: `http://freshmeat.sourceforge.net/projects/phccompiler`, *archived: url*
[6] *Roadsend PHP*: `http://roadsend.com/`, *archived: url*

by transforming PHP to Java bytecode. Their approach then relies on existing JVM implementations for further optimizations. These approaches are similar to *Source-to-Source* approaches but perform optimizations at runtime with a dynamic compiler. While such an approach saves engineering resources, the target runtime and its IR may not be optimized to run dynamically typed languages. This issue is referred to as "The Repurposed JIT Compiler Phenomenon" by Castanos et al. in [69]. To address the need to run dynamic languages on the JVM, the JSR 292 expert group introduced *invokedynamic*, a new bytecode instruction [70]. Nevertheless, Bolz et al. analyzed that hand-crafted JIT compilers still excel with better performance [71].

HHVM [3, 4] is a continuation of efforts put in *HPHPc*. It is a virtual machine with a state-of-the-art JIT compiler designed for Hack, a dialect of PHP. Some of its key optimization techniques include "type specialization, side exits, profile-guided optimizations, and region-based compilations". We refer to [72] for a more elaborate analysis on HHVM. HHVM dropped source compatibility for PHP after version v3.30 (end of 2019) [44].

HappyJIT and *HippyVM* [73, 74] are RPython/PyPy based PHP implementations. Similarly to Truffle, the PyPy project provides language agnostic tools to write a guest language as an interpreter in RPython, a dialect of Python. In contrast to Truffle and Graal, PyPy uses a meta-tracing compiler. For an elaborate analysis of tracing and partial evaluation, we refer to the works of Marr et. al in [75]. The PyPy project shows promising results. However, results in [73] were published in 2011 and we are unaware of PHP related follow-up work. Additionally, open source work on *HippyVM* seems to have stalled as the last commit dates back to 2015.

Methods we introduced in this section differ from *Source-to-Source* approaches by implementing or adapting a dynamic compiler. As previously explained, an approach which leverages dynamic compilation avoids challenges to model dynamic runtime behavior, such as *eval* or dynamic typing, because these optimization are now performed at runtime. Unlike other approaches, our work evaluates performance gains of a Truffle and Graal hosted implementation. Similarly to *HappyJIT* and *HippyVM* [73, 74], we model a guest language as an interpreter and use automatically derived optimizations to JIT compile the executed program. However, our approach is based on Graal and uses partial evaluation while PyPy uses tracing [75].

To the best of our knowledge, the most maintained and competitive runtimes for PHP are *Zend Engine*, *HHVM*, and *JPHP*. We benchmark these against our work on *graalphp*. More information to these implementations is presented in Section 4.1.3.

5.4 Truffle and GraalVM

Research on other language runtimes hosted on Truffle and GraalVM shows promising directions. Graal.JS (JavaScript) [76], TruffleRuby (Ruby) [77], and GraalPython (Python) [78] are examples of actively maintained dynamic language implementations hosted on Truffle. Their respective runtimes show competitive results compared to their reference implementations. We refer to the works of Gaikwad et al. for an elaborate performance analysis of languages hosted on Truffle [5].

Chapter 6

Future Work

In the limited scope of this thesis we implemented a subset of the language specification. Namely, we implemented language features imposed by three synthetic benchmarks. As of writing this report, the core-runtime of our language implementation reached c.a. 4000 LoC. We already discussed remaining language features in Section 3.4.

Graalphp-native

We provided a native build of *graalphp* but did not further optimize it. We believe that there is still much potential and interesting follow-up work in a native-build and integration with other native runtime libraries. In one of the benchmarks (Spectralnorm) *graalphp-native* exceeded peak performance of *graalphp* and all other implementations. PHP has a diverse ecosystem of runtime libraries reaching from image manipulations to database drivers. For user adoption these libraries must be integrated.

Runtime Libraries

An important aspect of a programming language is its runtime libraries. PHP has a variety of functions and third-party libraries written in C. We believe an approach that allows reusability of existing libraries is the most viable and future-proof attempt for adoption. GraalVM provides a native function interfaces to call native libraries. Future work on runtime libraries includes how existing native code can efficiently be invoked and how our implemented data types can be seamlessly shared with other libraries.

Web Server Integration

In order to evaluate performance of *graalphp* on real-world work load, we must implement server-side page generation features. We must evaluate

whether integration of our compiler into an existing web server such as Apache[1] is useful or whether we should build a new server or adapt existing JVM based solutions.

Truffle Language Interoperability

Truffle supports an interoperability feature to allow Truffle-hosted implementations to call into each other without overhead [79]. We intend to evaluate how interoperability can be integrated into our implementation.

[1] `https://httpd.apache.org/`, *archived: url*

Chapter 7

Conclusion

In this work we presented *graalphp*, the first experimental compiler and runtime for PHP hosted on Truffle and Graal. With our implemented feature set, we ran a selected number of synthetic benchmarks by *The Computer Language Benchmarks Game* [11]. Our experimental evaluation aimed to perform a preliminary performance comparison, given that the implementation is not yet fully realized. We implemented language features such as arrays, control-flow constructs, functions and datatypes. Our goal was to investigate potential performance gains of a Truffle-hosted implementation. The results have compared peak performance of *graalphp* with *Zend Engine*, PHP's reference implementation, as well as *HHVM*, *JPHP* and an early version of PHP 8. Our preliminary results suggest that a Truffle-hosted PHP implementation might be significantly faster than existing well-maintained language implementations. *Graalphp* reached a peak performance of up to 859% compared to *Zend Engine* and showed competitive to significantly higher performance gains compared to other implementations. While our preliminary results are promising, more engineering and research efforts must be invested into our implementation. For instance, for a more realistic work-load we aim to implement server-facing website rendering. Additionally, while an argument regarding code size is challenged to be representative in capturing complexity of a project, the core runtime of competitive projects are significantly larger. Zend Engine[1] for instance is over 120k lines of C code, while HHVM[2] exceeds 450k lines of cpp code. Our core runtime[3] is currently ca. 4000 lines of code and exceeds peak performance of other implementations on selected benchmarks. While our implementation is not fully realized, we developed it as open-source software [10] which was motivated to lay foundations for future work towards a feature complete and open implementation.

[1] Zend Engine: php-src/zend [80]
[2] HHVM: hphp/runtime [81]
[3] Graalphp: graalphp/graalphp-language [10]

Appendix A

Appendix

A.1 Implementing a Parser

In this section we document the decisions which led to graalphp-parser, our fork of the Eclipse PHP Development Tools (PDT). We compare existing parser for PHP source code and benchmark their performance.

A.1.1 Evaluation of Parsers

Migrating to another parser to a later extent in project development leads to unnecessary engineering efforts. Instantiating of Truffle nodes is dependent on how parsing results are curated. Hence, it is in our interest to carefully choose a parser. When opting for a parser for `graalphp` we prioritized criterions shown below;

1. Written in Java
2. Compatibility to the latest version of PHP
3. API to traverse parsing results
4. Parsing speed
5. Code quality and size of code base

There are a number of parsing project of PHP source code for the JVM. We compare ANTRL, JPHP and the Eclipse PHP Development Tools, some of the most mature and popular projects we found. All parsers support a recent version of PHP and are actively maintained.

ANTRL Parser

ANTLR, ANother Tool for Language Recognition, is a popular LL(*)[1] parser written in Java. It can generate lexers and parsers and supports a variety of source grammars including PHP [82]. Table A.1 summarizes opportunities and obstacles when adapting ANTRL.

Pros	Cons
Support for PHP 7.4	Slow Parsing speed (see benchmarks)
Grammar based parser	
Visitor pattern for AST traversal	
BSD 3-clause license	
Active community, 8k stars on Github	

Table A.1: Features of ANTRL parser

JPHP

JPHP is an alternative implementation of the PHP programming language on the JVM. It compiles PHP source code to Java bytecode. It implements a subset of the PHP runtime libraries and features own extensions [51]. The project includes a hand written parser. Features of JPHP are summarized in Table A.2.

Eclipse PHP Development Tools

The Eclipse PHP Development Tools is a distribution of the Eclipse IDE which supports development of PHP-based web applications. The project contains a Java CUP [83] and JFlex [84] based LALR[2] parser for PHP source code [37]. Features of the Eclipse PDT are listed in Table A.3.

Comparison

While ANTRL requires the least efforts to integrate in graalphp, it scales poorly for parsing large source files. The hand written parser used by the JPHP project is the fastest but is tightly integrated in the JPHP compiler. Furthermore, the lack of an AST traversal API makes it impractical to be

[1] Left-to-right, Leftmost derivation
[2] Look-Ahead Left-to-right, Rightmost derivation in reverse

Pros	Cons
Support for PHP 7.1+	No visitor pattern to traverse parsing results
Fast parsing speed (see benchmarks)	No clear separation between parser and compiler
Apache license 2.0	Hand written parsing logic (Token POJOs: 5000 LOC, Tokenizer: 1000 LOC, Parser: 4000 LOC)
Active community, 1.6k Stars on Github	Dependencies to non parsing dependent code (org.ow2.asm package)

Table A.2: Features of JPHP parser

Pros	Cons
Support for PHP 7.4	Tightly integrated in the Eclipse IDE with many dependencies
Grammar based parser	
AST visitor for tree transformations	
POJOs for all AST nodes	
Eclipse Public License 2.0	

Table A.3: Features of Eclipse PHP Development Tools

easily reused. Being written from scratch, adaptations to upcoming versions of PHP require more engineering efforts than a grammar based parser. The parser implemented by the Eclipse PHP Development Tools is deeply integrated in the Eclipse IDE, requiring additional efforts to modularize. The Eclipse parser provides POJOs[3] for AST nodes and traversal APIs to extract parsing results. As we see in benchmark section, it scales well with large source files. Extracting the parser used by the Eclipse project will presumably take less efforts than writing a new parser, or improving another parser. Based on these findings we provide a fork of the Eclipse PHP Development Tools. We remove all runtime libraries, rewrite the build system and provide a standalone JAR.

[3]POJO: Plain old Java object

A.1.2 graalphp-parser

Graalphp-parser is a fork of the Eclipse PHP Development Tools [37]. We removed all Eclipse IDE dependencies and provide it as a standalone parser supporting PHP 7.4. Our contributions entail the following work.

1. Remove dependencies to the Eclipse Dynamic Languages Toolkit and other parts of the Eclipse PHP Development Tools.
2. Rewrite the the build system to integrate Ant[4] artifacts into a Maven[5] release.
3. Provide the parser as a standalone Maven artifact.
4. Add better error handling and bail-out behavior.
5. Add test suites to compare parsing results of graalphp-parser with parsing results of Eclipse PHP Development Tools.

Graalphp-parser is released under the Eclipse Public License v2.0 and available as a Maven artifact. An example of parsing source code with graalphp-parser can be found in Listing 17.

[4] `https://ant.apache.org/`, *archived: url*
[5] `https://maven.apache.org/`, *archived: url*

```
ASTParser parser = ASTParser.newParser(PHPVersion.PHP7_4);
String msg = "I had a date() with PHP and I had to mktime() for it.";
String code = "<?php $a = '" + msg + "'; echo $a; ?>";
parser.setSource(code.toCharArray());
parser.addErrorListener(new ConsoleErrorListener()); // to stdout
parser.addErrorListener(new BailoutErrorListener()); // to exception
Program pgm = parser.parsePhpProgram();
```

```
<!-- pgm.toString() -->
<Program start='0' length='37'>
  <Statements>
    <ExpressionStatement start='6' length='19'>
      <Assignment start='6' length='18' operator='='>
        <Variable start='6' length='2' isDollared='true'>
          <Identifier start='7' length='1' name='a'/>
        </Variable>
        <Value>
          <Scalar start='11' length='13'
            type='string'
            value='&apos;I had a date() with PHP \
                                and had to mktime() for it.&apos;'/>
        </Value>
      </Assignment>
    </ExpressionStatement>
    <EchoStatement start='26' length='8'>
      <Variable start='31' length='2' isDollared='true'>
        <Identifier start='32' length='1' name='a'/>
      </Variable>
    </EchoStatement>
    <EmptyStatement start='35' length='2'/>
  </Statements>
  <Comments>
  </Comments>
</Program>
```

Listing 17: graalphp-parser API

A.1.3 Parsing Benchmarks

We benchmark parsing time of ANTLR, JPHP, and graalphp-parser for various sized source files. Results of these benchmarks were used to select a parser for graalphp. Benchmarks are performed with the Java Microbenchmark Harness[6] on JDK 11.0.7. The same hardware is used as declared in Section 4.1.4. We executed five warm-up and five measurement iterations of 10 seconds each. An overview of results is depicted in Figure A.1 and the tables A.4, A.5 and A.6 list measurements.

We notice that graalphp-parser is ca. 2x slower than the hand written parser in JPHP. For a file size of 8000 lines, this is a slow down of 25ms. We further see that ANTRL is c.a. up to two orders of magnitude slower than the fastest parser. This is likely due to its LL(*) look-ahead implementation. Repeated parsing of the same source file can further be optimized by a cache. The cache looks up a previously parsed result such that parsing of the same file can be avoided. This is not further discussed, however.

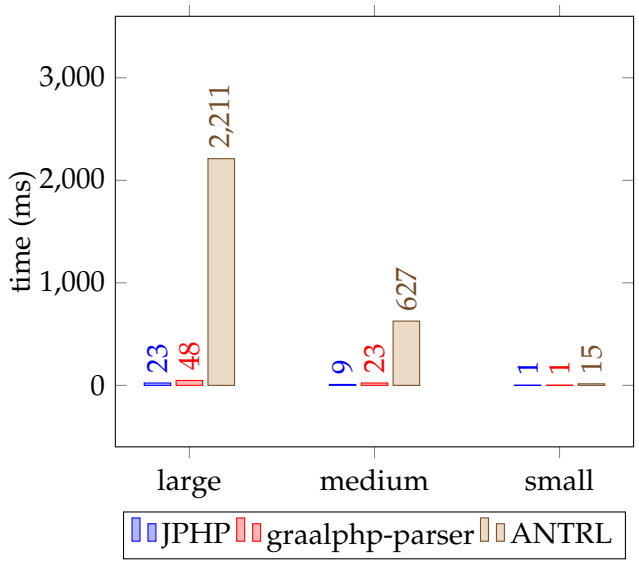

Figure A.1: Peak performance of parsing benchmark. Smaller is better. We parsed a large (8000 LOC), medium sized (2400 LOC) and small (50 LOC) file. The y-axis is peak execution time and the x-axis is the file size.

[6] `https://openjdk.java.net/projects/code-tools/jmh/`

Parser	Avg (ms)	Min (ms)	Max (ms)	Stdev
JPHP	22.619	22.581	22.713	0.053
Graalphp-parser	47.622	47.483	47.758	0.103
ANTRL	2210.715	2199.06	2225.774	11.264

Table A.4: Parsing large files (phpoffice/Xls.php), 7948 lines of code

Parser	Avg (ms)	Min (ms)	Max (ms)	Stdev
JPHP	8.821	8.814	8.827	0.006
Graalphp-parser	23.47	23.457	23.495	0.014
ANTRL	626.705	625.922	627.083	0.462

Table A.5: Parsing medium sized files (symfony/RequestTest.php), 2413 lines of code

Parser	Avg (ms)	Min (ms)	Max (ms)	Stdev
JPHP	0.652	0.644	0.659	0.006
Graalphp-parser	0.647	0.646	0.647	0.001
ANTRL	15.378	15.337	15.476	0.056

Table A.6: Parsing small files (benchmarkgame/fannkuchredux.php-1.php), 51 lines of code

Appendix B

Graalphp Source Assets

To reduce the size of this document we do not attach source assets. They can be downloaded at the following hyperlinks.

When developing *graalphp* during this thesis, we followed a vertical integration approach: features were written in git feature branches and were vertically integrated, from modifications in *graalphp-parser*, to *graaphp* to test suites. Continuous integration[1] ensured a stable master branch. The main repository currently contains more than 450 commits.

graalphp The main repository is hosted at [10]. It includes all *graalphp* runtime artifacts including *graalphp-parser*, Graal component and *graalphp-native* installer, as well as a *Dockerfile* and benchmark scripts.

graalphp-parser-benchmarks Evaluation scripts to benchmark *graalphp-parser* are hosted here[2].

eclipse-pdt-parser-fatjar A fat jar bundle of the Eclipse PDT parser is hosted here[3]. We used it to perform end-to-end testing against *graalphp-parser*. The fat jar ensures that *graalphp-parser* did not introduce semantic errors compared to its upstream version.

[1] `https://travis-ci.org/`

[2] `https://github.com/abertschi/graalphp-parser-benchmarks`

[3] `https://github.com/abertschi/eclipse-pdt-parser-fatjar`

Appendix C

Evaluation Assets

In this chapter we list benchmark files. To keep the size of this document within a reasonable margin we only show benchmark files for PHP and graalphp and refer to the source distribution for benchmarks on HHVM and JPHP.

C.1 Benchmark Source Files

C.1.1 Fannkuchredux

appendix/source-assets/fannkuchredux.php-1.php:

```
1  <?php
2  /* The Computer Language Benchmarks Game
3     https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4     contributed by Isaac Gouy, transliterated from Mike Pall's
   ↪  Lua program
5  */
6  // modifications to version on benchmarksgame website
7  // remove list() builtin and assign variables explicitly
8  // remove strings from printf and print result values directly
9  // use a fixed N instead of command line argument
10 // introduce warm up and timing

11 function Fannkuch($n){
12     $p = $q = $s = array();
13     $sign = 1; $maxflips = $sum = 0; $m = $n-1;
14     for ($i=0; $i<$n; $i++){ $p[$i] = $i; $q[$i] = $i; $s[$i] =
       ↪  $i; }
15     do {
```

```
16          // Copy and flip.
17          $q0 = $p[0];                                                    //
            ↪ Cache 0th element.
18          if ($q0 != 0){
19              for($i=1; $i<$n; $i++) $q[$i] = $p[$i];
                ↪ // Work on a copy.
20              $flips = 1;
21              do {
22                  $qq = $q[$q0];
23                  if ($qq == 0){
                    ↪ // ... until 0th element is 0.
24                      $sum += $sign*$flips;
25                      if ($flips > $maxflips) $maxflips = $flips;
                        ↪ // New maximum?
26                      break;
27                  }
28                  $q[$q0] = $q0;
29                  if ($q0 >= 3){
30                      $i = 1; $j = $q0 - 1;
31                      do { $t = $q[$i]; $q[$i] = $q[$j]; $q[$j] =
                        ↪ $t; $i++; $j--; } while ($i < $j);
32                  }
33                  $q0 = $qq; $flips++;
34              } while (true);
35          }
36          // Permute.
37          if ($sign == 1){
38              $t = $p[1]; $p[1] = $p[0]; $p[0] = $t; $sign = -1;
                ↪ // Rotate 0<-1.
39          } else {
40              $t = $p[1]; $p[1] = $p[2]; $p[2] = $t; $sign = 1;
                ↪ // Rotate 0<-1 and 0<-1<-2.
41              for($i=2; $i<$n; $i++){
42                  $sx = $s[$i];
43                  if ($sx != 0){ $s[$i] = $sx-1; break; }
44                  if ($i == $m) return array($sum,$maxflips);
                    ↪ // Out of permutations.
45                  $s[$i] = $i;
46                  // Rotate 0<-...<-i+1.
47                  $t = $p[0]; for($j=0; $j<=$i; $j++){ $p[$j] =
                    ↪ $p[$j+1]; } $p[$i+1] = $t;
48              }
49          }
50      } while (true);
```

```
51 }


52 $N = 12;
53 $iter = 15;

54 for($i = 0; $i < $iter; $i ++) {
55     $start=hrtime(true);
56     $A = Fannkuch($N);
57     $stop=hrtime(true);

58     $res = ($stop - $start) / 1000.0 / 1000.0;
59     output($N, $iter, $i, $res);
60     echo $A[0] . "\n";
61     echo $A[1] . "\n";

62 }

63 function output($N, $iters, $iter, $val) {
64     echo "fannkuchredux-php N/iters/iter/val;" . $N . ";" .
       ↪  $iters . ";" . $iter . ";" . $val . ";" . "\n";
65 }
66 ?>
```

Listing 18: fannkuchredux, PHP, fannkuchredux.php-1.php

appendix/source-assets/fannkuchredux.php-1.graalphp:

```
1  <?php
2  /* The Computer Language Benchmarks Game
3     https://salsa.debian.org/benchmarksgame-team/benchmarksgame/
4     contributed by Isaac Gouy, transliterated from Mike Pall's
   ↪  Lua program
5  */

6  // modifications to version on benchmarksgame website
7  // remove list() builtin and assign variables explicitly
8  // remove strings from printf and print result values directly
9  // use a fixed N instead of command line argument
10 // introduce warm up and timing

11 function Fannkuch($n){
12     $p = $q = $s = array();
13     $sign = 1; $maxflips = $sum = 0; $m = $n-1;
```

```
14      for ($i=0; $i<$n; $i++){ $p[$i] = $i; $q[$i] = $i; $s[$i] =
        ↪ $i; }
15      do {
16          // Copy and flip.
17          $q0 = $p[0];                                          //
            ↪ Cache 0th element.
18          if ($q0 != 0){
19              for($i=1; $i<$n; $i++) $q[$i] = $p[$i];
                ↪ // Work on a copy.
20              $flips = 1;
21              do {
22                  $qq = $q[$q0];
23                  if ($qq == 0){
                    ↪ // ... until 0th element is 0.
24                      $sum += $sign*$flips;
25                      if ($flips > $maxflips) $maxflips = $flips;
                        ↪ // New maximum?
26                      break;
27                  }
28                  $q[$q0] = $q0;
29                  if ($q0 >= 3){
30                      $i = 1; $j = $q0 - 1;
31                      do { $t = $q[$i]; $q[$i] = $q[$j]; $q[$j] =
                        ↪ $t; $i++; $j--; } while ($i < $j);
32                  }
33                  $q0 = $qq; $flips++;
34              } while (true);
35          }
36          // Permute.
37          if ($sign == 1){
38              $t = $p[1]; $p[1] = $p[0]; $p[0] = $t; $sign = -1;
                ↪ // Rotate 0<-1.
39          } else {
40              $t = $p[1]; $p[1] = $p[2]; $p[2] = $t; $sign = 1;
                ↪ // Rotate 0<-1 and 0<-1<-2.
41              for($i=2; $i<$n; $i++){
42                  $sx = $s[$i];
43                  if ($sx != 0){ $s[$i] = $sx-1; break; }
44                  if ($i == $m) return array($sum,$maxflips);
                    ↪ // Out of permutations.
45                  $s[$i] = $i;
46                  // Rotate 0<-...<-i+1.
47                  $t = $p[0]; for($j=0; $j<=$i; $j++){ $p[$j] =
                    ↪ $p[$j+1]; } $p[$i+1] = $t;
```

```
48              }
49          }
50      } while (true);
51  }


52  $N = 12;
53  $iter = 30;

54  for($i = 0; $i < $iter; $i ++) {
55      $start=graalphp_time_ns();
56      $A = Fannkuch($N);
57      $stop=graalphp_time_ns();

58      $res = ($stop - $start) / 1000.0 / 1000.0;
59      output($N, $iter, $i, $res);
60      println($A[0]); println($A[1]);
61  }

62  function output($N, $iters, $iter, $val) {
63      graalphp_print_args("fannkuchredux-gphp N/iters/iter/val",
        ↪  $N , $iters ,  $iter, $val);
64  }
65  ?>
```

Listing 19: fannkuchredux, graalphp, fannkuchredux.php-1.graalphp

C.1.2 Spectralnorm

`appendix/source-assets/spectralnorm.php-2-val.php:`

```
1  <?php
2  /* The Computer Language Benchmarks Game
3  https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4  contributed by Isaac Gouy
5  modified by anon
6  */

7  /*
8  modifications:
9   - pass integer by value not by reference
10  - pass global variable as argument instead of using global
    ↪  keyword
```

```
 - do not print additional text beside result value
 - replace for each by for keyword
 - add timing measurements
*/

// pass by val version passes global variable as argument by
↪  value instead of by reference

function A($i, $j){
    return 1.0 / ( ( ( ($i+$j) * ($i+$j+1) ) >> 1 ) + $i + 1 );
}

function Av($n, $v, $Av){
    for ($i = 0; $i < $n; ++$i) {
        $sum = 0.0;
        for($j = 0; $j < $n; $j++) {
            $v_j = $v[$j];
            $sum += A($i,$j) * $v_j;
        }
        $Av[$i] = $sum;
    }
    return $Av;
}

function Atv($n, $v, $Atv){
    for($i = 0; $i < $n; ++$i) {
        $sum = 0.0;
        for($j = 0; $j < $n; $j++) {
            $v_j = $v[$j];
            $sum += A($j,$i) * $v_j;
        }
        $Atv[$i] = $sum;
    }
    return $Atv;
}

function AtAv($n, $v, $_tpl){
    $tmp = Av($n,$v, $_tpl);
    return Atv($n, $tmp, $_tpl);
}

function doIteration($n) {
$u = array_fill(0, $n, 1.0);
$_tpl = array_fill(0, $n, 0.0);
```

```
48 for ($i=0; $i<10; $i++){
49     $v = AtAv($n,$u, $_tpl);
50     $u = AtAv($n,$v, $_tpl);
51 }

52 $vBv = 0.0;
53 $vv = 0.0;

54 for($i = 0; $i < $n; $i ++) {
55     $val = $v[$i];
56     $vBv += $u[$i]*$val;
57     $vv += $val*$val;
58 }
59 return sqrt($vBv/$vv);
60 }

61 $N = 5500;
62 $iter = 30;

63 for($i = 0; $i < $iter; $i ++) {
64     $start=hrtime(true);
65     $A = doIteration($N);
66     $stop=hrtime(true);

67     $res = ($stop - $start) / 1000.0 / 1000.0;
68     output($N, $iter, $i, $res);
69     echo $A . "\n";
70 }

71 function output($N, $iters, $iter, $val) {
72     echo "spectralnorm-val-php;" . $N . ";" . $iters . ";" .
       ↪  $iter . ";" . $val . ";" . "\n";
73 }
```

Listing 20: spectralnorm copy-by-value, PHP spectralnorm.php-2-val.php

`appendix/source-assets/spectralnorm.php-2-ref.php`:

```
1 <?php
2 /* The Computer Language Benchmarks Game
3 https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4 contributed by Isaac Gouy
5 modified by anon
```

```
6  */

7  /*
8  modifications:
9   - pass integer by value not by reference
10  - pass global variable as argument instead of using global
    ↪  keyword
11  - do not print additional text beside result value
12  - replace for each by for keyword
13  - add timing measurements
14 */

15 function A($i, $j){
16     return 1.0 / ( ( ( ($i+$j) * ($i+$j+1) ) >> 1 ) + $i + 1 );
17 }

18 function Av($n, &$v, &$_tpl){
19     $Av = $_tpl; // assign by value
20     for ($i = 0; $i < $n; ++$i) {
21         $sum = 0.0;
22         for($j = 0; $j < $n; $j++) {
23             $v_j = $v[$j];
24             $sum += A($i,$j) * $v_j;
25         }
26         $Av[$i] = $sum;
27     }
28     return $Av;
29 }

30 function Atv($n, &$v, &$_tpl){
31     $Atv = $_tpl;
32     for($i = 0; $i < $n; ++$i) {
33         $sum = 0.0;
34         for($j = 0; $j < $n; $j++) {
35             $v_j = $v[$j];
36             $sum += A($j,$i) * $v_j;
37         }
38         $Atv[$i] = $sum;
39     }
40     return $Atv;
41 }

42 function AtAv($n, &$v, &$_tpl){
43     $tmp = Av($n,$v, $_tpl);
```

```
44      return Atv($n, $tmp, $_tpl);
45  }

46  function doIteration($n) {
47  $u = array_fill(0, $n, 1.0);
48  $_tpl = array_fill(0, $n, 0.0);

49  for ($i=0; $i<10; $i++){
50      $v = AtAv($n,$u, $_tpl);
51      $u = AtAv($n,$v, $_tpl);
52  }

53  $vBv = 0.0;
54  $vv = 0.0;

55  for($i = 0; $i < $n; $i ++) {
56      $val = $v[$i];
57      $vBv += $u[$i]*$val;
58      $vv += $val*$val;
59  }
60  return sqrt($vBv/$vv);
61  }

62  $N = 5500;
63  $iter = 30;

64  for($i = 0; $i < $iter; $i ++) {
65      $start=hrtime(true);
66      $A = doIteration($N);
67      $stop=hrtime(true);

68      $res = ($stop - $start) / 1000.0 / 1000.0;
69      output($N, $iter, $i, $res);
70      echo $A . "\n";
71  }

72  function output($N, $iters, $iter, $val) {
73      echo "spectralnorm-ref-php;" . $N . ";" . $iters . ";" .
        ↪  $iter . ";" . $val . ";" . "\n";
74  }
```

Listing 21: spectralnorm copy-by-reference, PHP, spectralnorm.php-2-ref.php

`appendix/source-assets/spectralnorm.php-2-val.graalphp`:

```php
<?php
/* The Computer Language Benchmarks Game
https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

contributed by Isaac Gouy
modified by anon
*/

/*
modifications:
 - pass integer by value not by reference
 - pass global variable as argument instead of using global keyword
 - do not print additional text beside result value
 - replace for each by for keyword
*/

// pass by val version passes global variable as argument by value instead of by reference

function A($i, $j){
    return 1.0 / ( ( ( ($i+$j) * ($i+$j+1) ) >> 1 ) + $i + 1 );
}

function Av($n, $v, $Av){
    for ($i = 0; $i < $n; ++$i) {
        $sum = 0.0;
        for($j = 0; $j < $n; $j++) {
            $v_j = $v[$j];
            $sum += A($i,$j) * $v_j;
        }
        $Av[$i] = $sum;
    }
    return $Av;
}

function Atv($n, $v, $Atv){
    for($i = 0; $i < $n; ++$i) {
        $sum = 0.0;
        for($j = 0; $j < $n; $j++) {
            $v_j = $v[$j];
            $sum += A($j,$i) * $v_j;
        }
        $Atv[$i] = $sum;
```

```
37      }
38      return $Atv;
39  }

40  function AtAv($n, $v, $_tpl){
41      $tmp = Av($n,$v, $_tpl);
42      return Atv($n, $tmp, $_tpl);
43  }

44  function doIteration($n) {
45  $u = array_fill(0, $n, 1.0);
46  $_tpl = array_fill(0, $n, 0.0);

47  for ($i=0; $i<10; $i++){
48      $v = AtAv($n,$u, $_tpl);
49      $u = AtAv($n,$v, $_tpl);
50  }

51  $vBv = 0.0;
52  $vv = 0.0;

53  for($i = 0; $i < $n; $i ++) {
54      $val = $v[$i];
55      $vBv += $u[$i]*$val;
56      $vv += $val*$val;
57  }
58  return sqrt($vBv/$vv);
59  }

60  $N = 5500;
61  $iter = 50;

62  for($i = 0; $i < $iter; $i ++) {
63      $start=graalphp_time_ns();
64      $A = doIteration($N);
65      $stop=graalphp_time_ns();

66      $res = ($stop - $start) / 1000.0 / 1000.0;
67      output($N, $iter, $i, $res);
68      println($A);

69  }

70  function output($N, $iters, $iter, $val) {
```

```
71      graalphp_print_args("spectralnorm-val", $N , $iters ,
        ↪  $iter, $val);
72  }
```

Listing 22: spectralnorm copy-by-value, graalphp, spectralnorm.php-2-val.graalphp

appendix/source-assets/spectralnorm.php-2-ref.graalphp:

```
1   <?php
2   /* The Computer Language Benchmarks Game
3   https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4   contributed by Isaac Gouy
5   modified by anon
6   */

7   /*
8   modifications:
9    - pass integer by value not by reference
10   - pass global variable as argument instead of using global
    ↪  keyword
11   - do not print additional text beside result value
12   - replace for each by for keyword
13  */

14  function A($i, $j){
15      return 1.0 / ( ( ( ($i+$j) * ($i+$j+1) ) >> 1 ) + $i + 1 );
16  }

17  function Av($n, &$v, &$_tpl){
18      $Av = $_tpl; // assign by value
19      for ($i = 0; $i < $n; ++$i) {
20          $sum = 0.0;
21          for($j = 0; $j < $n; $j++) {
22              $v_j = $v[$j];
23              $sum += A($i,$j) * $v_j;
24          }
25          $Av[$i] = $sum;
26      }
27      return $Av;
28  }

29  function Atv($n, &$v, &$_tpl){
30      $Atv = $_tpl;
```

```
31      for($i = 0; $i < $n; ++$i) {
32          $sum = 0.0;
33          for($j = 0; $j < $n; $j++) {
34              $v_j = $v[$j];
35              $sum += A($j,$i) * $v_j;
36          }
37          $Atv[$i] = $sum;
38      }
39      return $Atv;
40  }

41  function AtAv($n, &$v, &$_tpl){
42      $tmp = Av($n,$v, $_tpl);
43      return Atv($n, $tmp, $_tpl);
44  }

45  function doIteration($n) {
46  $u = array_fill(0, $n, 1.0);
47  $_tpl = array_fill(0, $n, 0.0);

48  for ($i=0; $i<10; $i++){
49      $v = AtAv($n,$u, $_tpl);
50      $u = AtAv($n,$v, $_tpl);
51  }

52  $vBv = 0.0;
53  $vv = 0.0;

54  for($i = 0; $i < $n; $i ++) {
55      $val = $v[$i];
56      $vBv += $u[$i]*$val;
57      $vv += $val*$val;
58  }
59  return sqrt($vBv/$vv);
60  }

61  $N = 5500;
62  $iter = 50;

63  for($i = 0; $i < $iter; $i ++) {
64      $start=graalphp_time_ns();
65      $A = doIteration($N);
66      $stop=graalphp_time_ns();
```

```
67      $res = ($stop - $start) / 1000.0 / 1000.0;
68      output($N, $iter, $i, $res);
69      println($A);
70  }

71  function output($N, $iters, $iter, $val) {
72      graalphp_print_args("spectralnorm-ref-gphp", $N , $iters ,
        ↪  $iter, $val);
73  }
```

Listing 23: spectralnorm copy-by-reference, graalphp, spectralnorm.php-2-ref.graalphp

C.1.3 Binary-Trees

appendix/source-assets/binarytrees.php-3-val.php:

```
1   <?php

2   /* The Computer Language Benchmarks Game
3      https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4      contributed by Peter Baltruschat
5      modified by Levi Cameron
6   */
7   /*
8     moditifcations:
9     - Define command line arguments within script already (argv)
10    -  Simplified version of printf; print values without
    ↪  additional text
11    -  === operator (identity) replaced with == (equality)
12    - We run this benchmark twice. Once with explicit copy-by-ref
    ↪  semantics for
13    arrays (the & Operator), and one version which embodies
14    copy-by-value which is the default for PHP. By-reference
    ↪  results
15    show what a more sophisticated array implementation which may
16    implement copy-on-write or similar techniques can achieve.
17  */

18  function bottomUpTree($depth)
19  {
20     if (!$depth) return array(-1,-1);
21     $depth--;
```

```
22     return array(
23        bottomUpTree($depth),
24        bottomUpTree($depth));
25  }

26  function itemCheck($treeNode) {
27     return 1
28        + ($treeNode[0][0] == -1 ? 1 : itemCheck($treeNode[0]))
29        + ($treeNode[1][0] == -1 ? 1 : itemCheck($treeNode[1]));
30  }

31  function doAlgorithm($n) {
32      $minDepth = 4;
33      $maxDepth = max($minDepth + 2, $n);
34      $stretchDepth = $maxDepth + 1;

35      $stretchTree = bottomUpTree($stretchDepth);
36      echo $stretchDepth . "\n";
37      echo itemCheck($stretchTree) . "\n";

38      unset($stretchTree);

39      $longLivedTree = bottomUpTree($maxDepth);

40      $iterations = 1 << ($maxDepth);
41      do
42      {
43          $check = 0;
44          for($i = 1; $i <= $iterations; ++$i)
45          {
46              $t = bottomUpTree($minDepth);
47              $check += itemCheck($t);
48              unset($t);
49          }

50          echo $iterations . "\n";
51          echo $minDepth . "\n";
52          echo $check . "\n";


53          $minDepth += 2;
54          $iterations >>= 2;
55      }
56      while($minDepth <= $maxDepth);
```

```
57      echo $maxDepth . "\n";
58      echo itemCheck($longLivedTree) . "\n";

59  }
60  // benchmark

61  $N = 21;
62  $iter = 50;

63  for($i = 0; $i < $iter; $i ++) {
64      $start=hrtime(true);
65      doAlgorithm($N);
66      $stop=hrtime(true);

67      $res = ($stop - $start) / 1000.0 / 1000.0;
68      output($N, $iter, $i, $res);
69  }

70  function output($N, $iters, $iter, $val) {
71      echo "binary-trees-val N/iters/iter/val;" . $N . ";" .
        ↪  $iters . ";" . $iter . ";" . $val . ";" . "\n";
72  }

73  ?>
74
```

Listing 24: binary-trees copy-by-value, PHP binarytrees.php-3-val.php

appendix/source-assets/binarytrees.php-3-ref.php:

```
1   <?php

2   /* The Computer Language Benchmarks Game
3      https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4      contributed by Peter Baltruschat
5      modified by Levi Cameron
6   */
7   /*
8     moditifcations:
9     - Define command line arguments within script already (argv)
10    -  Simplified version of printf; print values without
    ↪  additional text
```

```
11    -  === operator (identity) replaced with == (equality)
12    - We run this benchmark twice. Once with explicit copy-by-ref
      ↪  semantics for
13    arrays (the & Operator), and one version which embodies
14    copy-by-value which is the default for PHP. By-reference
      ↪  results
15    show what a more sophisticated array implementation which may
16    implement copy-on-write or similar techniques can achieve.
17  */

18  // this version of the benchmark passes all arrays by reference
      ↪  where possible

19  function &bottomUpTree($depth)
20  {
21     if (!$depth) {
22     $A = array(-1,-1);
23     return $A;
24     }
25     $depth--;
26     $A = array(
27        bottomUpTree($depth),
28        bottomUpTree($depth));
29        return $A;
30  }

31  function itemCheck(&$treeNode) {
32     return 1
33        + ($treeNode[0][0] == -1 ? 1 : itemCheck($treeNode[0]))
34        + ($treeNode[1][0] == -1 ? 1 : itemCheck($treeNode[1]));
35  }

36  function doAlgorithm($n) {
37      $minDepth = 4;
38      $maxDepth = max($minDepth + 2, $n);
39      $stretchDepth = $maxDepth + 1;

40      $stretchTree = & bottomUpTree($stretchDepth);
41      echo $stretchDepth . "\n";
42      echo itemCheck($stretchTree) . "\n";

43      unset($stretchTree);

44      $longLivedTree = & bottomUpTree($maxDepth);
```

```
45      $iterations = 1 << ($maxDepth);
46      do
47      {
48          $check = 0;
49          for($i = 1; $i <= $iterations; ++$i)
50          {
51              $t = &bottomUpTree($minDepth);
52              $check += itemCheck($t);
53              unset($t);
54          }

55          echo $iterations . "\n";
56          echo $minDepth . "\n";
57          echo $check . "\n";


58          $minDepth += 2;
59          $iterations >>= 2;
60      }
61      while($minDepth <= $maxDepth);


62      echo $maxDepth . "\n";
63      echo itemCheck($longLivedTree) . "\n";

64  }
65  // benchmark

66  $N = 21;
67  $iter = 50;

68  for($i = 0; $i < $iter; $i ++) {
69      $start=hrtime(true);
70      doAlgorithm($N);
71      $stop=hrtime(true);

72      $res = ($stop - $start) / 1000.0 / 1000.0;
73      output($N, $iter, $i, $res);
74  }

75  function output($N, $iters, $iter, $val) {
76      echo "binary-trees-ref N/iters/iter/val;" . $N . ";" .
        ↪  $iters . ";" . $iter . ";" . $val . ";" . "\n";
```

```
77 }

78 ?>
79
```

Listing 25: binary-trees copy-by-reference, PHP, binarytrees.php-3-ref.php

`appendix/source-assets/binarytrees.php-3-val.graalphp:`

```
1  <?php
2  /* The Computer Language Benchmarks Game
3     https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4     contributed by Peter Baltruschat
5     modified by Levi Cameron
6  */
7  /*
8    moditifcations:
9    - Define command line arguments within script already (argv)
10   -  Simplified version of printf; print values without
   ↪  additional text
11   -  === operator (identity) replaced with == (equality)
12   - We run this benchmark twice. Once with explicit copy-by-ref
   ↪  semantics for
13   arrays (the & Operator), and one version which embodies
14   copy-by-value which is the default for PHP. By-reference
   ↪  results
15   show what a more sophisticated array implementation which may
16   implement copy-on-write or similar techniques can achieve.
17 */

18 function bottomUpTree($depth)
19 {
20    if (!$depth) return array(-1,-1);
21    $depth--;
22    return array(
23       bottomUpTree($depth),
24       bottomUpTree($depth));
25 }

26 function itemCheck($treeNode) {
27    return 1
28       + ($treeNode[0][0] == -1 ? 1 : itemCheck($treeNode[0]))
29       + ($treeNode[1][0] == -1 ? 1 : itemCheck($treeNode[1]));
30 }
```

```
31 function doAlgorithm($n) {
32     $minDepth = 4;
33     $maxDepth = max($minDepth + 2, $n);
34     $stretchDepth = $maxDepth + 1;

35     $stretchTree = bottomUpTree($stretchDepth);
36     println($stretchDepth);
37     println(itemCheck($stretchTree));

38     unset($stretchTree);

39     $longLivedTree = bottomUpTree($maxDepth);

40     $iterations = 1 << ($maxDepth);
41     do
42     {
43         $check = 0;
44         for($i = 1; $i <= $iterations; ++$i)
45         {
46             $t = bottomUpTree($minDepth);
47             $check += itemCheck($t);
48             unset($t);
49         }

50         println($iterations);
51         println($minDepth);
52         println($check);


53         $minDepth += 2;
54         $iterations >>= 2;
55     }
56     while($minDepth <= $maxDepth);


57     println($maxDepth);
58     println(itemCheck($longLivedTree));

59 }

60 $N = 21;
61 $iter = 15;
```

```
62 for($i = 0; $i < $iter; $i ++) {
63     $start=graalphp_time_ns();
64     doAlgorithm($N);
65     $stop=graalphp_time_ns();

66     $res = ($stop - $start) / 1000.0 / 1000.0;
67     output($N, $iter, $i, $res);
68 }

69 function output($N, $iters, $iter, $val) {
70     graalphp_print_args("binary-trees-val N/iters/iter/val", $N
       ↪  , $iters ,  $iter, $val);

71 }
72 ?>
```

Listing 26: binary-trees copy-by-value, graalphp, binarytrees.php-3-val.graalphp

appendix/source-assets/binarytrees.php-3-ref.graalphp:

```
1  <?php
2  /* The Computer Language Benchmarks Game
3     https://salsa.debian.org/benchmarksgame-team/benchmarksgame/

4     contributed by Peter Baltruschat
5     modified by Levi Cameron
6  */

7  /*
8    moditifcations:
9    - Define command line arguments within script already (argv)
10   -  Simplified version of printf; print values without
   ↪  additional text
11   -  === operator (identity) replaced with == (equality)
12   - We run this benchmark twice. Once with explicit copy-by-ref
   ↪  semantics for
13     arrays (the & Operator), and one version which embodies
14     copy-by-value which is the default for PHP. By-reference
   ↪  results
15     show what a more sophisticated array implementation which
   ↪  may
16     implement copy-on-write or similar techniques can achieve.
17 */
```

```
18 // this version of the benchmark passes all arrays by reference
   ↪ where possible

19 function &bottomUpTree($depth)
20 {
21    if (!$depth) return array(-1,-1);
22    $depth--;
23    return array(
24       bottomUpTree($depth),
25       bottomUpTree($depth));
26 }

27 function itemCheck(&$treeNode) {
28    return 1
29       + ($treeNode[0][0] == -1 ? 1 : itemCheck($treeNode[0]))
30       + ($treeNode[1][0] == -1 ? 1 : itemCheck($treeNode[1]));
31 }

32 function doAlgorithm($n) {
33     $minDepth = 4;
34     $maxDepth = max($minDepth + 2, $n);
35     $stretchDepth = $maxDepth + 1;

36     $stretchTree = &bottomUpTree($stretchDepth);
37     println($stretchDepth);
38     println(itemCheck($stretchTree));

39     unset($stretchTree);

40     $longLivedTree = &bottomUpTree($maxDepth);

41     $iterations = 1 << ($maxDepth);
42     do
43     {
44         $check = 0;
45         for($i = 1; $i <= $iterations; ++$i)
46         {
47             $t = & bottomUpTree($minDepth);
48             $check += itemCheck($t);
49             unset($t);
50         }

51         println($iterations);
52         println($minDepth);
```

```
53          println($check);


54          $minDepth += 2;
55          $iterations >>= 2;
56      }
57      while($minDepth <= $maxDepth);


58      println($maxDepth);
59      println(itemCheck($longLivedTree));

60  }

61  $N = 21;
62  $iter = 50;

63  for($i = 0; $i < $iter; $i ++) {
64      $start=graalphp_time_ns();
65      doAlgorithm($N);
66      $stop=graalphp_time_ns();

67      $res = ($stop - $start) / 1000.0 / 1000.0;
68      output($N, $iter, $i, $res);
69  }

70  function output($N, $iters, $iter, $val) {
71      graalphp_print_args("binary-trees-ref N/iters/iter/val", $N
        ↪  , $iters ,  $iter, $val);

72  }
73  ?>
```

Listing 27: binary-trees copy-by-reference, graalphp, binarytrees.php-3-ref.graalphp

Appendix D

Acknowledgment

The author of this thesis would like to thank Zhendong Su and Manuel Rigger from ETH Zurich for their supervision on this thesis. In particular, Manuel Rigger for his insightful advice throughout the project. We would also like to thank Christian Humer from Oracle Labs for his code review on the array implementation.

Bibliography

[1] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1916–1923, New York, NY, USA, 2009. Association for Computing Machinery.

[2] 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. *SIGPLAN Not.*, 47(10):575–586, October 2012.

[3] Facebook. HHVM, the HipHop Virtual Machine. `https://hhvm.com`, *archived: url*. Accessed on 20-08-03.

[4] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. *SIGPLAN Not.*, 49(10):777–790, October 2014.

[5] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. Performance analysis for languages hosted on the truffle framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Tiobe Software B.V. TIOBE programming community index. `https://www.tiobe.com/tiobe-index/`, *archived: url*, Accessed: 20-08-15.

[7] Pierre Carbonnelle. PYPL PopularitY of Programming Language. `https://pypl.github.io/PYPL.html`, *archived: url*, Accessed: 20-08-15.

[8] RedMonk. The RedMonk Programming Language Rankings: June 2020. https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/, *archived: url*, Accessed: 20-08-15.

[9] W3Techs, Q-Success Web-based Services. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php, Accessed: 20-08-15.

[10] Andrin Bertschi. Graalphp, A PHP implementation built on GraalVM. https://github.com/abertschi/graalphp, *archived: url*. Accessed on 20-08-28.

[11] Doug Bagley, Brent Fulgham, and Isaac Gouy. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html, *archived: url*, 2004. Accessed on 20-07-08.

[12] PHP. Language Specification for PHP. https://github.com/php/php-langspec, *archived: url*. Accessed 20-08-10.

[13] PHP. The History of PHP. https://www.php.net/manual/en/history.php.php, *archived: url*. Accessed 20-08-10.

[14] PHP. Introduction of Language Specification for PHP. https://news-web.php.net/php.internals/75886, *archived: url*. Accessed 20-08-10.

[15] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 683–702, New York, NY, USA, 2012. Association for Computing Machinery.

[16] PHP. Strict Types in PHP. https://www.php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration.strict, *archived: url*, Accessed: 20-08-10.

[17] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 944–953, New York, NY, USA, 2006. Association for Computing Machinery.

[18] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *ICOOOLPS@ECOOP*, 2009.

[19] M Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[20] Carl Friedrich Bolz, Armin Rigo. How to Not Write Virtual Machines for Dynamic Languages. In 3rd Workshop on Dynamic Languages and Applications, 2007.

[21] OpenJDK, John Rose. JEP 243: Java-Level JVM Compiler Interface. `https://openjdk.java.net/jeps/243`, *archived: url*, 2014-10-29. Accessed on 20-07-23.

[22] Google. V8 JavaScript Engine. `https://v8.dev/`, *archived: url*. Accessed on 20-07-23.

[23] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, David Cox, Thomas Rodriguez, and Kenneth Russell. Design of the java hotspot tm client compiler for java 6. *ACM Transactions on Architecture and Code Optimization*, 2008.

[24] PyPy. PyPy. `https://www.pypy.org/`, *archived: url*. Accessed 20-08-10.

[25] OpenJDK, Igor Veresov. JEP 317: Experimental Java-Based JIT Compiler. `https://openjdk.java.net/jeps/317`, *archived: url*. Accessed on 20-07-23.

[26] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir : An extensible declarative intermediate representation. 2013. In Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.

[28] Oracle. GraalVM Native Image, Reference Manual. `https://www.graalvm.org/docs/reference-manual/native-image`, *archived: url*. Accessed on 20-08-03.

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[30] OpenJDK. Hotspot Performance Tactics. `https://wiki.openjdk.java.net/display/HotSpot/PerformanceTacticIndex`, *archived: url*. Accessed 20-08-10.

[31] PHP. PHP RFC: JIT. `https://wiki.php.net/rfc/jit`, *archived: url*, Accessed: 20-09-10.

[32] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.

[33] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. *SIGPLAN Not.*, 48(2):73–82, October 2012.

[34] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. Supporting on-stack replacement in unstructured languages by loop reconstruction and extraction. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery.

[36] PHP. Supported Versions. `https://www.php.net/supported-versions.php`, *archived: url*, Accessed: 20-08-18.

[37] Eclipse Foundation. Eclipse PHP Development Tools. `https://www.eclipse.org/pdt/`, *archived: url*. Accessed on 20-07-23.

[38] PHP. Supported Datatypes in PHP. `https://www.php.net/manual/en/language.types.intro.php`, *archived: url*, Accessed: 20-08-18.

[39] PHP. Ternary Operator Associativity. `https://wiki.php.net/rfc/ternary_associativity`, *archived: url*, Accessed: 20-08-18.

[40] Stefan Marr and Benoit Daloze. Few versatile vs. many specialized collections: How to design a collection library for exploratory programming? In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion,

page 135–143, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. *SIGPLAN Not.*, 48(10):167–182, October 2013.

[42] Oracle. Truffle Libraries. `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/library/package-summary.html`, *archived: url*, Accessed: 20-09-01.

[43] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, page 133–144, New York, NY, USA, 2014. Association for Computing Machinery.

[44] HHVM. Ending Support for PHP. `https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html`, *archived: url*, Accessed: 20-08-28.

[45] The Computer Language Benchmarks Game, Fannkuchredux. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/fannkuchredux-php-1.html`, *archived: url*. Accessed on 20-07-08.

[46] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[47] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[48] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, page 63–74, New York, NY, USA, 2013. Association for Computing Machinery.

[49] The Computer Language Benchmarks Game, Binary Trees. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/binarytrees-php-3.html`, *archived: url*. Accessed on 20-07-08.

[50] The Computer Language Benchmarks Game, Spectralnorm. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/spectralnorm-php-2.html`, *archived: url*. Accessed on 20-07-08.

[51] JPHP, an alternative to PHP on the JVM. `http://jphp.develnext.org/`, *archived: url*. Accessed on 20-07-23.

[52] PHP. PHP Manual, References. `https://www.php.net/manual/en/language.references.php`, *archived: url*. Accessed on 20-08-03.

[53] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. Supporting on-stack replacement in unstructured languages by loop reconstruction and extraction. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing low-level languages to the jvm: Efficient execution of llvm ir on truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2016, page 6–15, New York, NY, USA, 2016. Association for Computing Machinery.

[55] Salim S. Salim, Andy Nisbet, and Mikel Luján. Trufflewasm: A webassembly interpreter on graalvm. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 88–100, New York, NY, USA, 2020. Association for Computing Machinery.

[56] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, page 3, USA, 2010. USENIX Association.

[57] 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, page 575–586, New York, NY, USA, 2012. Association for Computing Machinery.

[58] Nikita Popov. PHP 7 Virtual Machine. https://nikic.github.io/2017/04/14/PHP-7-Virtual-machine.html, *archived: url*, Accessed: 20-08-14.

[59] Nikita Popov. Internal value representation in PHP 7. https://nikic.github.io/2015/05/05/Internal-value-representation-in-PHP-7-part-1.html, *archived: url*, Accessed: 20-08-15.

[60] Julien Pauli, Nikita Popov, Anthony Ferrara. PHP Internals Book, Basic structure. http://www.phpinternalsbook.com/php7/internal_types/zvals/basic_structure.html, *archived: url*, Accessed: 20-08-15.

[61] Rohit Kulkarni, Aditi Chavan, and A Hardik. Transpiler and it's advantages. *International Journal of Computer Science and Information Technologies*, 6(2):1629–1631, 2015.

[62] Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., USA, 1995. https://stroustrup.com/dne.html, *archived: url*, Accessed: 20-08-10.

[63] Microsoft. The TypeScript Language. https://www.typescriptlang.org/, *archived: url*, Accessed: 20-08-10.

[64] Roadsend PHP. Roadsend PHP compiler/implementation. https://github.com/weyrick/roadsend-php, *archived: url*, Accessed: 20-08-10.

[65] Chaitali Tambe, Pramod Pawar, Dashrath Mane. PHP Optimization Using Hip Hop Virtual Machine. In *International Journal of Advanced Research in Computer Engineering and Technology(IJARCET), Volume 4 Issue 6*, June 2015.

[66] P. Biggar. Design and implementation of an ahead-of-time compiler for PHP. 2010. Trinity College (Dublin Ireland).

[67] Michiaki Tatsubori, Akihiko Tozawa, Toyotaro Suzumura, Scott Trent, and Tamiya Onodera. Evaluation of a just-in-time compiler retrofitted for php. *SIGPLAN Not.*, 45(7):121–132, March 2010.

[68] Caucho Technology, Resin. Quercus. https://www.caucho.com/resin-3.1/doc/quercus.xtp, *archived: url*, Accessed: 20-08-14.

[69] Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. *SIGPLAN Not.*, 47(10):195–212, October 2012.

[70] Christian Thalinger and John Rose. Optimizing invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, page 1–9, New York, NY, USA, 2010. Association for Computing Machinery.

[71] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98:408 – 421, 2015. Special Issue on Advances in Dynamic Languages.

[72] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 151–165, New York, NY, USA, 2018. Association for Computing Machinery.

[73] Homescu, Andrei and Şuhan, Alex. HappyJIT: a tracing JIT compiler for PHP. *DLS'11 - Proceedings of the 7th Symposium on Dynamic Languages*, 01 2011.

[74] HippyVM. HippyVM. `https://github.com/hippyvm/hippyvm`, *archived: url*, Accessed: 20-08-14.

[75] Stefan Marr and Stéphane Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 821–839, New York, NY, USA, 2015. Association for Computing Machinery.

[76] Oracle. Graal.JS. `https://github.com/graalvm/graaljs`, *archived: url*, Accessed: 20-08-14.

[77] Oracle. TruffleRuby. `https://github.com/oracle/truffleruby`, *archived: url*, Accessed: 20-08-14.

[78] Oracle. Graalpython. `https://github.com/graalvm/graalpython/`, *archived: url*, Accessed: 20-08-14.

[79] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. *SIGPLAN Not.*, 51(2):78–90, October 2015.

[80] PHP. PHP source distribution of Zend Engine. `https://github.com/php/php-src`, *archived: url*, Accessed: 20-09-01.

[81] Facebook. HHVM source distribution. `https://github.com/facebook/hhvm`, *archived: url*, Accessed: 20-09-01.

[82] Terence Parr. ANother Tool for Language Recognition. `https://www.antlr.org/`, *archived: url*. Accessed on 20-07-23.

[83] Hudson S. CUP Parser Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`, *archived: url*, Princeton University, 1999.

[84] Gerwin Klein. JFlex. `http://jflex.de`, *archived: url*. Accessed on 20-07-23.

Andrin **Bertschi**

Software Engineer

abertschi
andrinbertschi
https://abertschi.ch
hi@abertschi.ch
Zurich, CH

Education

ETH Zurich **Zurich, ZH**
B.Sc. Computer Science *2016 - 2020*

Thurgauerisch-Schaffhauserische Maturitätsschule für Erwachsene **Frauenfeld, TG**
High school for Adults, Passerelle *2015 - 2016*

Lehre Informatiker Applikationsentwicklung EFZ[1] **Winterthur, ZH**
Berufsfachschule Winterthur *2010 - 2014*

Work Experience

AXA **Winterthur, ZH**
Title: Java Engineer *08/2015 - 07/2016*
Java Enterprise backend development for insurance applications

AXA **Winterthur, ZH**
Title: Junior Java Engineer, Member of Competence Center Java *10/2014 - 07/2015*
Java Enterprise and web development

AXA Winterthur and AXA Technology Services Switzerland AG **Winterthur, ZH**
Title: Software Engineering Apprentice *08/2010 - 07/2014*
Mainframe application development on z/OS ISPF with PL/1, DB2 (2010-2012)
Backend development in Java EE (2012-2014)

Miscellaneous

Skills: Self-Reliance, Eager to Learn, Analytical Thinking, Team-Player

Areas of Interest: Compilers, Operating Systems, Program Analysis, Android

Languages: German (native), English (C1-C2), French (A1)

Non-Technical Hobbies: Powerlifting, Dancing

[1] Diploma Swiss Federal Vocational Education and Training in Software Development