# Using Procedure Cloning for Performance Optimization of Compiled Dynamic Languages

Robert Husák[1][a], Jan Kofroň[1][b], Jakub Míšek[2][c] and Filip Zavoral[1][d]
[1]*Faculty of Mathematics and Physics, Charles University, Ke Karlovu 3, Prague, Czech Republic*
[2]*iolevel s.r.o., Zelená 1743, Prague, Czech Republic*

Keywords: Compilers, Dynamic Languages, Optimization, Procedure Cloning, Type Analysis.

Abstract: Compilation of dynamic programming languages into strongly typed platforms such as .NET and JVM has proven useful in enhancing the security and interoperability of the resulting programs, as well as in enabling their source-less distribution. In order to produce the best intermediate code possible, dynamic language compilers can employ advanced interprocedural type analysis and perform various optimizing transformations. However, these efforts are often limited due to the ambiguity of types in these languages and the need to preserve soundness. In this paper, we improve the performance of global functions by adapting the technique of procedure cloning, focusing on different parameter types those specialized clones can be called with. We devise several heuristics to identify the most promising specializations and call them from their respective call sites. Our technique was implemented into PeachPie, a compiler of PHP to .NET, and evaluated on standard benchmarks distributed with PHP. Benchmarks containing deep recursion show a speedup factor up to 3.65, while benchmarks of computation-intensive loops reveal a speedup factor up to 2.64.

## 1 INTRODUCTION

Dynamic programming languages have become highly popular, mainly for their gentle learning curve and vast community support. The typical way to execute them is by using a runtime environment specialized for a particular programming language, such as Zend Engine[1] for PHP and V8[2] for JavaScript. However, there have been several successful attempts in their ahead-of-time (AOT) compilation to intermediate languages of strongly typed platforms such as .NET or Java (Míšek and Zavoral, 2019). This approach enhances their security, enables source-less distribution, and simplifies integration with other programming languages on the respective platforms.

This kind of AOT compilation has its own unique benefits and challenges in performance optimization. As both .NET and Java runtimes are equipped with a highly optimized just-in-time (JIT) compiler and their intermediate representations are stack-based, an AOT compiler of a dynamic language does not need to perform low-level optimizations such as common subexpression elimination or copy propagation. On the other hand, the AOT compiler cannot directly introduce low-level runtime optimizations of its own, specialized for the particular dynamic language. Furthermore, runtime optimizations on these platforms are usually aimed at strongly-typed object-oriented languages instead of dynamic languages with dynamic typing. For example, their JIT compilers lack the ability to specialize compiled code according to the object types actually occurring in program execution, which is a common feature of runtimes designed for dynamic languages (Chambers and Ungar, 1991; Ottoni, 2018; Chevalier-Boisvert and Feeley, 2014; Chevalier-Boisvert et al., 2010).

As a result, the optimization effort of AOT dynamic language compilers targeting .NET or Java is limited to producing the most type-specific and efficient intermediate code possible. The main problem arises from how dynamic languages treat variables, as the types of their values can change during variable lifetimes and differ from one execution to another. While careful flow-sensitive type analysis can identify variable types in some cases (Míšek et al., 2016),

[a] https://orcid.org/0000-0001-7128-6163
[b] https://orcid.org/0000-0003-0391-4812
[c] https://orcid.org/0000-0002-0792-2054
[d] https://orcid.org/0000-0003-3140-8538
[1] https://www.php.net
[2] https://v8.dev

Husák, R., Kofroň, J., Míšek, J. and Zavoral, F.
Using Procedure Cloning for Performance Optimization of Compiled Dynamic Languages.
DOI: 10.5220/0011272300003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 175-186
ISBN: 978-989-758-588-3; ISSN: 2184-2833

its precision is limited by the need to stay conservative and not allow any transformations which would possibly introduce runtime errors. One of the main sources of this imprecision is the parameters of global functions because the compiler must assume that each function can possibly be called with arguments of any type. As a result, any operations on parameters must be accompanied by type checks, causing significant overhead, especially in deeply recursive functions and computation-intensive loops. This is the problem we decided to tackle in this work, and we explain it further in section 2.

Our contributions are as follows:

- In section 3, we describe the overall idea of using procedure cloning with specialized parameter types and how it integrates into a dynamic language compiler.
- Section 4 explains different approaches to generate procedure clones specialized by their parameter types.
- Selecting the suitable targets of call sites is described in section 5.
- We measure the efficiency of our optimization and its different variants in section 6.

Section 7 describes the most relevant work related to our research, while section 8 concludes.

## 2 PROBLEM

To illustrate the challenges faced by AOT dynamic language compilers to strongly-typed platforms, we use PHP code and its compilation to .NET using the compiler PeachPie (Míšek and Zavoral, 2019). While certain aspects of the problem might be unique for both PHP and PeachPie, the fundamental issue of dynamic typing stands for other languages and runtimes as well.

The current state-of-the-art type analysis during the compilation is shown on the PHP code in Figure 1 and the C# equivalent of the Common Intermediate Language (CIL) produced by PeachPie (Míšek et al., 2016) in Figure 2a[3].

As we can see, the three global PHP functions `foo`, `bar` and `baz` were compiled into the corresponding methods in .NET. All of them have `PhpValue` as their return type, which is a union-like structure containing a value of any possible type. It must be used whenever there is any uncertainty about the type of the particular value. While `PhpValue` preserves the expected semantics, any operations later performed on it cause additional overhead, because the exact value type must be checked during runtime. Furthermore, returning more precise types than `PhpValue` can help to better propagate type information among the program.

Another overhead stems from how PHP handles assignments of strings and associative arrays. Although both structures are allocated on the heap and can be large, assigning a value of one of these types into a variable has a copy-by-value semantics. The same behavior takes place when such an argument is passed to a function. For example, if `foo` passed an array to `baz` and its content was modified in there through the parameter `$a`, these changes must not be propagated back to the array referenced by the variable `$r2` in `foo`. The same behavior is expected for strings, as they are mutable in PHP. Because a parameter of the type `PhpValue` can, among other things, contain a string or an array, a call to `PhpValue.Copy` is made to ensure that the copy-by-value semantics is preserved.

Notice that `bar` and `baz` use `PhpValue` for their parameters, whereas `foo` uses more specific ones, namely `int` and `PhpArray`, because its parameters are enhanced with type declarations. Although `a` still has to be copied explicitly, it can use `PhpArray.Copy` to bypass the type check. Moreover, we were able to skip the copying of `i`, because integers are passed by value in .NET.

Because parameter type declarations are optional and `bar` and `baz` do not use them, they had to be compiled with no particular types. Therefore, we must convert all the arguments passed to them on their call sites in `foo` to `PhpValue`, losing compile-time type information and postponing the decisions based on it to runtime. For example, `bar` must dynamically find the implementation of the operator `+` using `PhpValue.Add`, as it can represent both a number addition and an array union.

As we can see, the overhead associated with calling `bar` exceeds its actual semantics by a great amount. If `bar` is called several million times in a loop or as a part of a deeply recursive function, most of the computation time will be spent on type checking inside the methods of `PhpValue`.

Nevertheless, there are multiple opportunities to simplify the produced code and make it more efficient. Thanks to the type analysis, we know which types are passed as arguments to both `bar` and `baz`. Therefore, we can create their overloads with specialized types and call them from the respective call sites in `foo`, as depicted in Figure 2b. The original overloads with unspecified parameter types are not

[3]Note that we simplified certain implementation details and renamed several library functions to enhance clarity.

```
function foo(int $i, array $a) {
  $r1 = bar($i, 4);
  $r2 = bar($a, []);
  $r3 = baz($r2);
  return $r3;
}

function bar($x, $y) {
  return $x + $y;
}
```

```
function baz($a) {
  if (is_array($a)) {
    return count($a);
  } else {
    return $a;
  }
}
```

Figure 1: A sample PHP code to showcase the potential improvements of interprocedural type analysis.

```
PhpValue foo(int i, PhpArray a)
{
  a = PhpArray.Copy(a);
  PhpValue r1 =
    bar((PhpValue)i, (PhpValue)4);
  PhpValue r2 =
    bar((PhpValue)a,
        (PhpValue)new PhpArray());
  PhpValue r3 =
    baz((PhpValue)r2);
  return r3;
}

PhpValue bar(PhpValue x, PhpValue y)
{
  x = PhpValue.Copy(x);
  y = PhpValue.Copy(y);
  return PhpValue.Add(x, y);
}

PhpValue baz(PhpValue a)
{
  a = PhpValue.Copy(a);
  if (a.IsArray)
    return (PhpValue)Variables.count(a);
  else
    return a;
}
```

(a) The result of the current state-of-the-art technique, without routine cloning.

```
int foo(int i, PhpArray a)
{
  a = PhpArray.Copy(a);
  PhpNumber r1 = bar(i, 4);
  PhpArray r2 = bar(a, new PhpArray());
  PhpValue r3 = baz(r2);
  return r3;
}

PhpNumber bar(int x, int y)
{
  return PhpNumber.Add(x, y);
}

PhpArray bar(PhpArray x, PhpArray y)
{
  x = PhpArray.Copy(x);
  y = PhpArray.Copy(y);
  return PhpArray.Union(x, y);
}

int baz(PhpArray a)
{
  a = PhpArray.Copy(a);
  return Variables.count(a);
}
```

(b) The result after the application of procedure cloning with parameter type specialization.

Figure 2: The C# representation of the CIL compiled from the PHP code in Figure 1.

displayed to save space, but they must be defined as well, in case they were called dynamically, e.g., using call_user_func.

The first specialized overload of bar expects the parameters x and y to be integers. As a result, the compiler could eliminate their initial copying, select the proper addition operator, and find a more specialized return type. In PHP, the sum of two integers becomes a floating-point number if it overflows the integer range; therefore, the specialized structure PhpNumber is used. Although not as efficient as int, its overhead is still lower than the one of PhpValue.

The second specialized overload of bar receives arrays as its parameters, enabling the compiler to eliminate the type checks used in the operations of PhpValue, and select the proper operation of PhpArray.Union for the + operator. Furthermore, the return type of the overload has been made more precise, as it must certainly be an array. Note that the enhanced precision of return types is critical, as the information gained from it can spread throughout the code.

Furthermore, because r2 in foo becomes an instance of PhpArray, passing it to baz creates its specialized overload. In this case, the branch condition is_array is evaluated to true, discarding the else branch. This specializes the return type of baz to int, propagating to the return type of foo as well.

```
ASTs ← ParseSyntaxTrees()
CFGs ← BindCFGs(ASTs)
Analyse(CFGs)
while CFGs changed and maxIters not reached do
  specCFGs ← CreateClones(CFGs)
  Analyse(specCFGs)
  CFGs ← CFGs ∪ specCFGs
  SpecializeCallSites(CFGs)
  Transform(CFGs)
  Analyse(CFGs)
Emit(CFGs)
```

Figure 3: Overall pipeline of a dynamic language compiler enhanced with procedure cloning.

The particular technique selected in the example produces procedure clones according to the argument types passed at call sites. While this approach is natural, there are multiple other options worth exploring. For example, the compiler can inspect the code of the cloned procedure itself and search for indications about possible parameter types. Comparing the most promising procedure cloning techniques is one of the main contributions of this paper.

The following section describes the general structure of the solution and how it can be integrated into a dynamic language compiler.

## 3 ARCHITECTURE

In Figure 3, we explain how to incorporate procedure cloning into a pipeline of an existing AOT dynamic language compiler. In the beginning, the compiler converts a set of abstract syntax trees (`ASTs`) into a set of control flow graphs (`CFGs`). All the other operations are performed directly on `CFGs`. This approach has proven to work well in PeachPie, and we will use it for the purpose of this explanation. Other ways are possible, too, such as annotating and transforming `ASTs` directly.

Let us now focus on the details of our pipeline shown in Figure 3. `Analyse` performs a data-flow analysis (Aho et al., 2006) to discover the possible types of each expression in the program. The analysis is flow-sensitive and interprocedural, although it only propagates return types from callees to callers, not the other way around. The analysis creates a call graph, connecting call sites with all the possible callees as a byproduct. Details of this analysis are explained in (Míšek and Zavoral, 2017). There is a special value representing any possible type, which is used, e.g., for parameters without type hints, values read from global variables, arrays, possibly aliased variables, and procedures containing `eval`. While PeachPie supports dynamic code evaluation, the most efficient results can be obtained when most of the code is provided at compile-time, enabling the utilization of the aforementioned analysis properly.

The following part of the algorithm in Figure 3 consists of standard analysis and optimizing transformation loop, as known from traditional compilers (Aho et al., 2006). All the information gathered by `Analyse` can be then used by `Transform` to simplify the program while preserving its semantics, e.g., by evaluating type checking functions in branch conditions and removing unreachable branches. The whole mechanism is further explained in (Husák et al., 2020).

The main contribution of this paper lies in the functions `CreateClones` and `SpecializeCallSites`, executed right after the first analysis round. `CreateClones` is responsible for selecting the procedures to be cloned and choosing the parameter types of these clones. Before adding them to `CFGs`, `Analyse` is executed on them so that they obtain the necessary type annotations. `SpecializeCallSites` inspects each call site targeting a cloned procedure and selects the clones with the most fitting parameter types. To preserve soundness, the set of these clones must always fully cover all the type combinations of the parameters possibly passed to it at the given call site.

To illustrate the need of repeating the analysis, specialization, and transformation in a loop, let us review the sample PHP code in Figure 1 and the desired result of its compilation in Figure 2b. Assume the most straightforward implementations of the aforementioned functions: `CreateClones` simply traverses all call sites and produces the specialized clones according to the parameter types, while `SpecializeCallSites` always selects only the clone with the perfect parameter match and resorts to the original general definition otherwise. During the first iteration, two specialized overloads of `bar` are produced and used from their respective call sites. However, `baz` cannot be specialized yet, because the precise type of `$r2` is not revealed until the next invocation of `Analysis`. The second call to `CreateClones` finally produces the desired specialized overload, and its return type is propagated to the return type of `foo` in the subsequent analysis of all CFGs.

Due to the simplicity of our example, we were able to create specialized clones exactly matching the parameter types of all call sites. However, in real-world applications, we may not be able to achieve it due to the inability to properly identify the argument types the given procedure is called with. As a result, we can resort to the best-effort approach—we select

```
/**
 * @param int|string      $x
 * @param bool|null|array $y
 */
function foo($x, $y) {
  if ($x) {
    echo $y[$x];
  } else if (is_null($y)) {
    echo "nothing";
  }
  // ...
}
```

Figure 4: A sample PHP code with PHPDoc type annotation.

the clones to enable the best optimizations possible within the given procedure and let the call sites handle the selection of the appropriate clone at runtime. This situation occurs when SpecializeCallSites selects more than one clone for a particular call site. Later, the call site is emitted as an if-else statement, selecting the appropriate specialized clone by the parameter types. We must carefully consider when to apply this strategy, as the added overhead on the call site could eventually revert the benefit of the optimizations performed inside the called procedure. The task to find the desired balance is described in the following two sections, as they provide the particular implementations of CreateClones and SpecializeCallSites, respectively.

## 4 SPECIALIZED CLONE GENERATION

As the situations where our approach might be used vary greatly, we decided to create a set of different techniques for clone creation and pinpoint their potential strong and weak points. Each technique has total freedom over the parameter types it chooses for the specialization without the risk of breaking the soundness of the program, as explained in section 5. Our work covers only the specialization of global functions, but it can be extended to cover class methods as well. The list of techniques follows:

**PhpDoc:** Before delving into techniques applicable to any general situation, let us at first focus on long-lived well-maintained projects whose authors often write parameter type information inside structured comments. A format called PHPDoc is used in the case of PHP; we can see its example in Figure 4. Although we could compile such annotated functions directly into a strongly-typed code, this can change the code semantics by introducing additional type checks and conversions, which is usually not wanted. Instead, we can use our approach of specialized clone creation, possibly improving the performance while preserving the semantics. If each parameter has at most one type annotation, selecting the type for the particular parameter is straightforward. However, in our example, we need to create all the combinations to maximize the chances of being called with specialized parameters, hence six clones in total. If the comment is outdated from the actual implementation and foo in our example never receives an integer as the first argument, three clones are produced unnecessarily. Furthermore, the specialization of an argument to a particular type may not even benefit from the specialized routine in terms of more efficient code.

**ParamUsage:** Instead of depending on the type annotations from developers, we can estimate the parameter types automatically. In our example in Figure 4, we can see that $y is indexed as an array and later it is checked for its equality with null. From this information, we can infer that $y can be an array or null. Similarly, we can inspect $x and then produce all the clones as all the combinations of the corresponding types. The complete list of patterns recognized in a routine body for a parameter $p with the corresponding specialized types is shown in Table 1. The function type is used to obtain the precise type of the expression E, previously inferred by type analysis. The function containing scans all the available classes for those containing all the fields and methods accessed through the parameter $p. To prevent an uncontrollable growth of routine clones, should more than four classes be conforming to those criteria, only System.Object is selected, being the base class for all reference types in .NET. Notice the distinction between System.String and PhpString. While the former is an immutable UTF-16 string commonly used by all .NET languages, the latter is a PeachPie runtime class developed to match the semantics of mutable PHP strings. As a result, the primary target for optimization is System.String, as thanks to its immutability, it is not necessary to copy it upon every assignment, unlike PhpString.

The main benefit of the approach of *ParamUsage* is that we choose the types so that the particular specialized clone has a significant chance to be optimized later. For example, specializing the type of $y to array causes a simpler operation to be generated for $y[$x] and the removal of the is_null check.

**CallSites:** The previous techniques can suffer from the fact that although the specialized clones may be well-optimized, we have no information as to whether they will indeed be called, and the optimization will be worth the increased compilation time. A

Table 1: Patterns recognized for type specialization of parameters in the technique *ParamUsage*.

| Pattern | Type of parameter `$p` |
|---|---|
| `is_bool($p)` | `bool` |
| `is_int($p)` | `int` |
| `is_float($p)` | `float` |
| `$p instanceof T` | `T` |
| `$p[*]`, `is_array($p)` | `PhpArray` |
| `$p . *` | `System.String` |
| `is_string($p)` | `System.String`, `PhpString` |
| `$p === E` | `type(E)` |
| `is_null($p)` | `null` |
| `$p->F`, `$p->M(*)` | `containing(F, M)` |

straightforward solution to this problem is to traverse all the known call sites of the given routine and generate the specialized clones according to the types we know are passed as arguments. Note that by repeatedly running the analysis and transformation, we can even discover new specializations of already specialized routines.

**Targeted:** *ParamUsage* is not concerned about the context in which a procedure is called and *CallSites* does not take into account its actual contents. Therefore, they may both create a large number of specialized clones which will never be used in runtime or will not provide interesting performance benefits. To alleviate the problem while utilizing the information gathered by these techniques, we can combine them. *Targeted* selects the intersection of *ParamUsage* and *CallSites*, efficiently creating clones only when they have a chance to optimize the generated code and, at the same time, have high potential to be called. The downside of this technique is that the exact type combinations may not meet, creating too few clones.

**CallSitesSimplified:** Another way to reduce the number of clones while retaining as most gathered information as possible is to transform the specializations suggested by any of the previous techniques. To demonstrate this approach, we will use *CallSites* as the basis, and for each set of clones of a particular routine, we will create at most two clones. The first clone is considered the “best-case scenario” while the second one will be the “worst-case scenario”. As an example, consider the following three suggested specialized clones of a function `bar`:

- `bar(int, PhpValue, System.String)`
- `bar(PhpValue, PhpArray, System.String)`
- `bar(int, PhpString, System.String)`

An important thing to mention is that not all specialized parameter types bring the same benefits. For example, simple types such as `null`, `bool`, `int` and `float` can reduce a lot of unnecessary overhead by skipping copying the value at the start of the function. On the other hand, specialization to `PhpArray` only enables to skip certain type checks. As a result, we can assign a distinct priority to each type, according to our demand on specializing it in clones:

1. `null`
2. `bool`
3. `int`
4. `float`
5. `PhpNumber` (an union of `int` and `float`)
6. `System.String` (.NET UTF-16 immutable string)
7. `System.Object` (.NET base class for reference types)
8. All user-defined classes
9. `PhpString`
10. `PhpArray`
11. `PhpValue` (an union of all possible types)

Given this ordering, we can select the best and the worst type for each parameter in an easy way, yielding:

- `bar(int, PhpString, System.String)`
- `bar(PhpValue, PhpValue, System.String)`

The first clone ensures that we do not miss suitable optimization opportunities, whereas the second one is used to optimize this function for its most common use. In this example, the function `bar` is always called with a string constant as its last parameter, while the types of the first and the second parameter vary.

## 5 CALL SITE TARGET SELECTION

The previous section explained how a set of specialized clones for each procedure is produced. Now we focus on the process happening at each call site where one of the specialized routines is called, showing the mechanism of how to select the appropriate clone.

We start by ordering all the clones of the given procedure according to their expected performance benefits. As a sufficient approximation, we use the priorities mentioned in the previous section in the technique *CallSitesSimplified* and sort the clones in a lexicographic manner by their parameters. For example, the clones of the function `bar` used in the recent example will be ordered as follows:

1. `bar(int, PhpString, System.String)`
2. `bar(int, PhpValue, System.String)`
3. `bar(PhpValue, PhpArray, System.String)`

The following part is unique for each call site, as it takes the particular arguments into account. Thanks to the previous type analysis, we know the upper bound of types of all the arguments passed to the call; therefore, we can inspect their compatibility with the specialized parameters. For each argument and parameter, there are three result options:

- *Always:* The argument can always be passed to the parameter, e.g., a subclass to its base class.
- *RuntimeDependent:* The argument can be passed, but needs a prior type check at runtime, such as `PhpValue` to `int`.
- *Never:* The argument can never be passed, e.g., an integer to an array.

First, we filter out all the clones where any of the arguments can *never* be passed to its corresponding parameter, removing unfeasible specializations. If no clone is left, the attempt for specialization is abandoned, and the original unspecialized procedure is called. Otherwise, given the aforementioned ordering, we search for the first clone where all the arguments can *always* be passed to their corresponding parameters. If such a clone is found, the call site is updated to call the clone instead of the original procedure, providing a more specialized and potentially more efficient program without runtime overhead. In the case when no such a clone exists, we distinguish between two different approaches:

**Branch:** The first of the feasible clones is selected for an opportunistic type check at runtime, potentially being the one with the highest potential performance benefit. As a result, the call site is extended to an `if-else` branch, and the specialized clone is called if the types match; otherwise, the original definition is called. Although our framework allows for more than one specialization at the call site, the experiments in the next section show that one runtime specialization already brings a substantial performance benefit.

**Static:** Since unsuccessful runtime type checks cause additional overhead without any performance benefits, this approach does not allow them to be performed. As a result, if a specialized clone whose parameters can be *always* assigned from the arguments is not found, the original unspecialized routine is called.

## 6 EVALUATION

This section aims to validate our approach on real-world examples. We will focus both on the solution as a whole and the differences between the particular techniques. The research questions are as follows:

- **RQ1:** *What is the performance impact on different kinds of programs?*
- **RQ2:** *How significant is the compilation overhead in terms of time and resulting assembly size?*
- **RQ3:** *What are the strengths and weaknesses of different techniques?*

For this purpose, we have implemented our techniques into a modified version of PeachPie (Míšek and Zavoral, 2019) and allowed each one of them to be turned on during compilation. The first scenario consists of the official PHP benchmarks[4], originally used to measure the performance of Zend Engine. Among others, they contain the computations of the Ackermann function, the Fibonacci numbers, matrix multiplication, and a deeply nested loop. We skipped the computation of the Mandelbrot set as it does not take any parameters; therefore, our optimization approach cannot be applied. Furthermore, several short functions with almost immeasurable run time were not considered, either. The benchmark results are shown by the particular specialization technique in Figure 5[5]. The table shows an average of 8 runs of each benchmark, preceded by 4 warm-up runs. The name of each technique is enhanced by the *-Static* or *-Branch* suffix to distinguish between their different variants of call site target selection.

The second scenario simulates sequential requests to a front page of a website implemented in WordPress 5.6.0. The code consists of about 248 KLOC in 933 PHP files; the count of called procedures during each request is 28,800. To measure performance reliably, we employ the BenchmarkDotNet[6] library. For each specialization technique, the compiled WordPress website is requested 300 times to warm up, and the same number of requests is then performed for the actual measurement. The results are listed in Table 3, using the same naming convention as before. Apart from the performance, we are interested in other statistics as well, e.g., the compilation time, the number of produced specialized overloads, and

[4]https://github.com/php/php-src/blob/master/Zend/bench.php

[5]The experiments were conducted on a desktop equipped with an AMD Ryzen 9 3900X 12-Core CPU and 32GB RAM.

[6]https://benchmarkdotnet.org

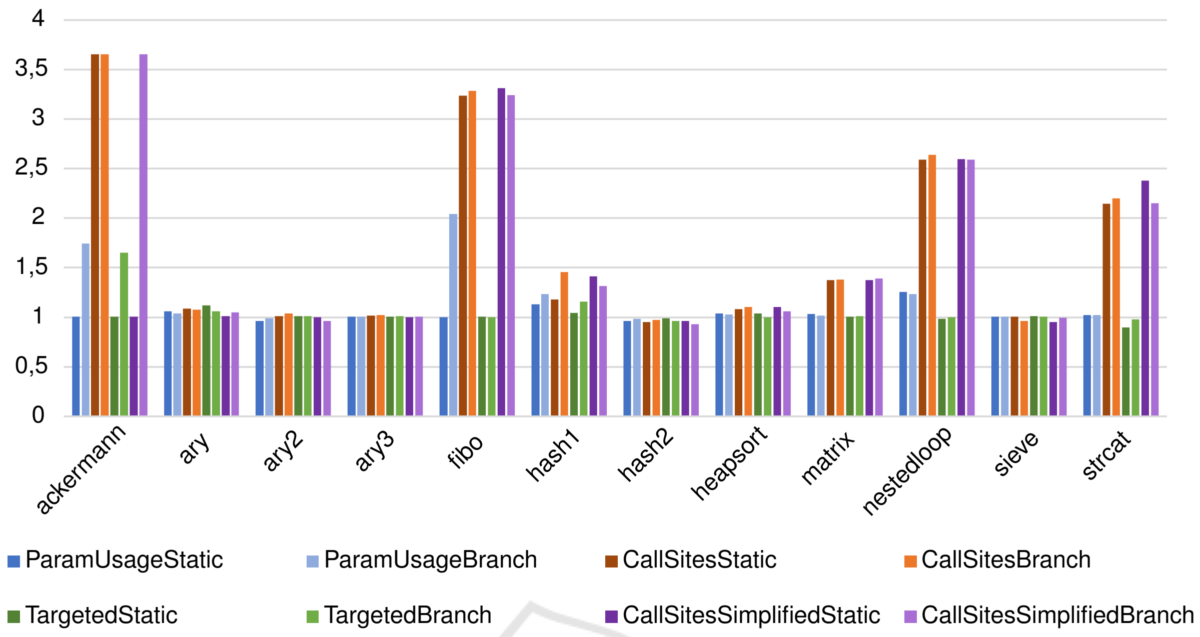

Figure 5: Results of the official PHP benchmarks compiled using PeachPie enhanced with our techniques for specialized overload generation. Each column shows the speedup factor of the given technique against the original program without specializations.

Table 2: WordPress compilation and front page generation statistics.

| Technique | Compilation | Assembly | Specialized | Calls | |
|---|---|---|---|---|---|
| | time | size | clones | Unspecialized | Specialized |
| Original | 19.12 s | 17,527 kB | 0 | 0 | 0 |
| PhpDocOverloadsStatic | 33.47 s | 20,840 kB | 3,314 | 22,687 | 4,321 |
| PhpDocOverloadsBranch | 29.79 s | 21,191 kB | 3,314 | 13,327 | 13,681 |
| ParamUsageStatic | 28.93 s | 22,159 kB | 3,059 | 7,976 | 2,840 |
| ParamUsageBranch | 28.94 s | 22,370 kB | 3,059 | 1,472 | 9,344 |
| CallSitesStatic | 33.89 s | 20,315 kB | 2,487 | 19,222 | 6,610 |
| CallSitesBranch | 34.71 s | 20,489 kB | 2,484 | 12,165 | 13,667 |
| TargetedStatic | 23.78 s | 18,101 kB | 319 | 3,109 | 2,224 |
| TargetedBranch | 23.62 s | 18,153 kB | 319 | 526 | 4,807 |
| CallSitesSimplifiedStatic | 23.12 s | 19,304 kB | 1,517 | 20,464 | 5,268 |
| CallSitesSimplifiedBranch | 23.57 s | 19,506 kB | 1,517 | 12,475 | 13,257 |

| Technique | Branched calls | | |
|---|---|---|---|
| | Total | Selecting original | Selecting specialized clone |
| PhpDocOverloadsBranch | 4,799 | 485 | 4,314 |
| ParamUsageBranch | 6,800 | 311 | 6,489 |
| CallSitesBranch | 3,512 | 901 | 2,611 |
| TargetedBranch | 2,616 | 40 | 2,576 |
| CallSitesSimplifiedBranch | 4,966 | 993 | 3,973 |

Table 3: Results of the WordPress front page generation benchmark measured using BenchmarkDotNet for particular techniques. Speedup factor of mean time and relative memory savings (higher is better) are displayed.

| Technique | Time | Memory |
|---|---|---|
| Original | 66.4 ms | 19.3 MB |
| PhpDocOverloadsStatic | 1.003 | 1.003 |
| PhpDocOverloadsBranch | 1.019 | 1.032 |
| ParamUsageStatic | 1.009 | 1.003 |
| ParamUsageBranch | 1.017 | 1.032 |
| CallSitesStatic | 1.017 | 1.005 |
| CallSitesBranch | 1.025 | 1.038 |
| TargetedStatic | 0.998 | 1.002 |
| TargetedBranch | 1.015 | 1.024 |
| CallSitesSimplifiedBranch | 1.006 | 1.024 |
| CallSitesSimplifiedStatic | 0.996 | 1.003 |

the resulting assembly size. Other interesting runtime statistics include, e.g., the number of specialized overload calls. To gather them, we performed a separate single front page request with specially instrumented code. All the statistics are shown in Table 2. The instrumentation also served to trace the calls of all the procedures and their argument values. Those traces were all equal to the trace of the original version without specializations, ensuring that none of the techniques changed the observable semantics of the original program.

The purpose of all these benchmarks is to evaluate the impact of procedure cloning techniques in the context of an AOT PHP compiler. The original PHP runtime is not included in the evaluation because its comparison with PeachPie is not within the scope of this paper. We refer anyone interested in this topic to the existing publications regarding PeachPie itself (Míšek and Zavoral, 2019; Míšek and Zavoral, 2017; Míšek et al., 2016) and the recent run of TechEmpower Web Framework Benchmarks[7].

Based on the presented data, we can answer the research questions:

**RQ1:** Regarding computation-intensive tasks, according to Figure 5, our techniques can achieve significant speedup factors when applied to the official PHP benchmarks. If we focus on the individual benchmarks, the highest speedup factor of 3.65 is reached by computing the Ackermann function for values $(3,7)$, while the second-highest speedup factor 3.31 holds the computation of the 30th Fibonacci number. Both computations are intentionally implemented as a very high number of recursive calls; therefore, even a slight overhead reduction in each call leads to a massive improvement of the total time. The third most improved benchmark is `nestedloop`, with the speedup factor of 2.64. Although the function is called only once in the last case, it performs more than 4.8 million comparisons during its execution. Originally, each comparison was required to check the operand types before selecting the proper operation. In the specialized versions, simple integer comparisons are performed instead, significantly lowering the overhead. Although the results show scarce cases of slowdowns, all of them are limited to reasonable bounds of several percent of performance.

According to the amount of reduced overhead, the improvements of particular benchmarks vary. The WordPress scenario recorded in Table 3 shows that the impact of our techniques on the computation time and memory consumption is positive but insignificant in comparison with the previous scenario. Several factors contribute to this situation. First, in WordPress, functions are often called by a dynamic name, not known at compile-time; therefore, our approach cannot analyze them properly. Second, we may not have enough information to create proper function specializations due to obtaining function arguments from locations without sufficient type annotation (e.g., arrays and fields). Third, our techniques currently specialize global functions only, while WordPress uses custom classes as well. Last, much work is performed at places that we cannot improve with our techniques, such as database communication, array operations, and string concatenation. Nevertheless, even though our techniques have not improved performance significantly, they have not worsened it, either. Therefore, the presented solution can be safely used on larger projects with a chance to improve certain parts but without the risk of decreasing their performance.

**RQ2:** Table 2 shows that the overhead of WordPress compilation time ranges from 21% to 81%, while the assembly size increases within 3% and 28%, depending on the particular technique. Understandably, the assembly size correlates with the number of specialized overloads produced, and the techniques using branched call sites produce larger assemblies than their static counterparts.

Because the overload specialization is expected to be done only during a build performed before publishing the particular website into a production environment, we find the compilation overhead level reasonable and not limiting for the developers. If a project is sensitive to assembly size or compilation time, a technique limiting the number of produced overloads must be selected, such as *CallSitesSimplifiedBranch*.

**RQ3:** The evaluation confirms our concerns ex-

[7]https://www.techempower.com/benchmarks

pressed for the particular techniques in section 4. Before inspecting the particular techniques, let us review the differences between the *Static* and *Branch* variants in general. As mentioned before, the first expected difference is a slight assembly size increase in the case of the *Branch* variant, caused by more verbose call sites. Performance-wise, *Branch* variant tends to amplify the ability of the given technique to choose the particular argument types correctly. Good choices are awarded a higher ratio of specialized overload selections, while wrong guesses only cause additional overhead of extra type checking. Generally speaking, the *Static* variant presents a conservative approach with less chance to improve the performance but with no risk of decreasing it. The *Branch* variant adds a slight risk of performance decrease, but the potential increase tends to be much more significant, as seen in Figure 5. Furthermore, the last three columns in Table 2 show that the branched calls were successful in calling specialized overloads in most cases.

Let us now focus on the differences between the particular techniques. As WordPress is well-documented, *PhpDoc* works reasonably well there, producing the highest number of specialized overloads and calling the highest number of them during execution, thus saving resources. However, we cannot use it for the PHP benchmarks because they do not contain any PHPDoc comments.

*ParamUsage* produces a similar number of specialized overloads as *PhpDoc*, but they are called by about 32% less often. Nevertheless, it gives WordPress the same performance boost as *PHPDoc*. However, this technique cannot exploit all the optimization potential in the PHP benchmarks because it lacks the information about which argument types are actually used in the functions calls.

On the other hand, *CallSites* focuses on precisely this problem. De facto creating a dedicated specialization for each call site reaches the best performance boosts in both scenarios. The disadvantage of this technique is that its compilation takes the longest time.

*Targeted* attempts to combine the best features of the last two mentioned techniques by producing only the intersection of the specialized overloads produced by them. The problem with this approach is that although both original sets are large, the resulting intersection is too small. Therefore, it misses most optimization opportunities in the PHP benchmarks.

Finally, *CallSitesSimplified* presents a reasonable compromise between the compilation time, the size of the resulting assembly, the number of specialized overloads, and the efficiency of the produced code. Although it may miss specific optimization opportunities, the resulting performance is still one of the highest, while the compilation overhead is one of the lowest.

As a result, we recommend the *Branch* variant of *CallSites* as the default setting, while the other options may serve for possible fine-tuning according to a particular scenario. For example, *CallSitesSimplified* can be useful to reduce compilation time and assembly size.

## 7 RELATED WORK

The idea of procedure cloning and specialization is well-known from the history of compiler research. Cooper et al. (Cooper et al., 1993) suggested the usage of procedure cloning within a Fortran compiler, enabling more context-sensitive interprocedural optimizations. Chambers and Ungar (Chambers and Ungar, 1991) suggested using method specializations in object-oriented language runtimes to alleviate the overhead of dynamic dispatch. This work was later extended by Dean et al. (Dean et al., 1995). In their algorithm, a profiling phase is performed to discover frequently called method chains. This data is then used to specialize methods according to the types of their parameters, effectively hoisting dynamic dispatches from frequently executed methods to less frequently executed ones. The main contribution of our solution is that we apply procedure cloning in the context of an AOT PHP compiler without any access to profiling data. According to the experience of Dean et al., creating specialized procedure clones based on the Cartesian product of possible parameter types does not cause an uncontrollable growth of procedure count. We can confirm this observation for PHP as well.

This work directly builds upon the research related to compilers from PHP to .NET: Phalanger (Benda et al., 2006; Abonyi et al., 2009), and its successor PeachPie (Míšek and Zavoral, 2010; Míšek et al., 2016; Míšek and Zavoral, 2019). PeachPie already contains a sophisticated interprocedural type analysis (Míšek and Zavoral, 2017) and a code transformation phase (Husák et al., 2020). However, the type information could spread only from callees to callers via return types. Procedure cloning presented in this work enables the utilization of the type information available in call sites to produce more efficient specialized clones of callees.

There are several other examples of dynamic language compilers which produce intermediate representation targeting existing managed runtimes for strongly-typed languages. JPHP[8] and Quercus[9] compile PHP to Java bytecode. Rhino JavaScript compiler[10] targets JVM as well, whereas Jurassic[11] produces .NET assemblies. We have not found any evidence about the usage of interprocedural optimization techniques within these compilers, as their documentations mainly focus on interoperability with other languages.

Another category of dynamic language compilers includes those directly producing native (executable) code, possibly utilizing a C/C++ compiler. In order to maximize the compatibility with the original PHP interpreter, phc (Biggar et al., 2009; Biggar, 2010) is strongly interconnected with Zend Engine, enabling it, e.g., to call existing PHP extensions directly. Prior to compilation, phc performs a sophisticated full-program analysis involving context-sensitive alias analysis (Emami et al., 1994), type analysis, and heap abstraction. As a result, programs compiled with phc can be optimized better than with our approach, but the scalability of the analysis is limited, especially considering that it cannot compile just a part of the solution and needs the entire code in advance. Furthermore, the analysis is entirely skipped, e.g., when encountering an `eval` statement. The HipHop compiler for PHP (Zhao et al., 2012) does not support dynamic code evaluation as well; for type inference, it uses Damas-Milner constraint-based algorithm (Damas and Milner, 1982), discovering specific types where possible. Hopc (Serrano, 2018) compiles JavaScript, targeting mainly embedded devices. Alongside a sound type analysis based on occurrence typing (Tobin-Hochstadt and Felleisen, 2010), the most likely types for all function parameters are collected, and its specialized version is created. The same feature is provided for MATLAB in the AOT component of MaJIC (Almási and Padua, 2002). Both of these approaches are similar to our *ParamUsage* technique with the difference that Hopc and MaJIC produce only one clone for each function rather than creating a clone for each possible combination of expected parameter types. Furthermore, other techniques presented in this paper also consider the types of arguments the functions are called with, potentially capturing more opportunities for optimization. Our most significant contribution concerning Hopc and MaJIC is that we introduced procedure cloning into the compilation of PHP, showing its specific challenges and optimization opportunities.

While our work is oriented on AOT compilation, the mainstream way to execute programs written in dynamic languages is customized runtime environments, usually equipped with JIT compilers. By having access to the actual runtime values and the control of the entire execution, they can afford to perform more aggressive and speculative optimizations than an AOT compiler. HHVM (Ottoni, 2018), runtime for the language Hack, specializes regions of CFG blocks by the types of used program variables. The decision about which blocks to specialize and for which types is performed based on the information gathered using profiling. Higgs JavaScript virtual machine (Chevalier-Boisvert and Feeley, 2014) uses a technique called *lazy basic block versioning*, where each CFG block is compiled only when the execution reaches it. JIT compiler then specializes it for the actual variable types obtained in the execution. McVM MATLAB virtual machine (Chevalier-Boisvert et al., 2010) compiles a specialized version of a function for each parameter type combination passed to it at runtime. Other examples of highly optimized runtime environments for dynamic languages include V8 JavaScript engine, PyPy (Rigo and Pedroni, 2006), and GraalVM with Truffle framework (Wimmer and Würthinger, 2012). Although our work can draw inspiration from specific technical solutions in the aforementioned virtual machines, their techniques are hard to use in the AOT compilation context, as they usually depend strongly on the actual runtime behavior. An AOT compiler cannot afford to compile only the functions and CFG blocks that will be actually executed without making sure that there is a fallback in case any optimistic assumption proves not to be true. Despite these challenges, moving specialization to the level of individual blocks within a CFG might be an interesting research direction for AOT dynamic language compilers.

## 8 CONCLUSION

In this paper, we have presented a set of techniques for dynamic language compiler optimization. The techniques incorporate procedure cloning into the compilation workflow, utilizing existing interprocedural type analysis to determine specialized parameter types. Important parts of the solution are the heuristics which ensure the specialized clones to be produced only in appropriate cases, not adding unnecessary overhead to the compilation nor runtime.

We implemented the techniques into the PeachPie compiler and evaluated their impact on standard PHP benchmarks. The results have shown that the highest execution time optimization can be reached in deeply

[8]https://github.com/jphp-group/jphp
[9]http://quercus.caucho.com
[10]https://github.com/mozilla/rhino
[11]https://github.com/paulbartrum/jurassic

recursive functions, e.g., the Ackermann function of $(3,7)$ was computed 3.65 times faster; the 30th Fibonacci number was computed 3.31 times faster. The speedup factor of computation-intensive loops ranges from 1.24 to 2.64. Although such significant improvements could not be achieved in real-world applications, validation on a WordPress website has shown the safety of our optimizations, as they never worsen the performance nor change the program semantics.

In our future work, we plan to extend the specialization mechanism even further and improve its effect on real-life websites, e.g., by producing clones not only of global functions but of class methods as well. Moreover, procedure inlining will be explored as another possible context-sensitive interprocedural optimization in dynamic language AOT compilers.

## ACKNOWLEDGEMENTS

This work has been supported by the Czech Science Foundation project no. 20-07487S, Charles University Grant Agency (GA UK) project no. 896120, the project PROGRESS Q48 and the grant SVV-260588.

## REFERENCES

Abonyi, A., Balas, D., Beňo, M., Míšek, J., and Zavoral, F. (2009). Phalanger improvements. Technical report, Charles University in Prague.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman.

Almási, G. and Padua, D. (2002). MaJIC: Compiling MATLAB for speed and responsiveness. *SIGPLAN Not.*, 37(5):294–303.

Benda, J., Matousek, T., and Prosek, L. (2006). Phalanger: Compiling and running PHP applications on the Microsoft .NET platform. *.NET Technologies 2006*.

Biggar, P. (2010). *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin.

Biggar, P., de Vries, E., and Gregg, D. (2009). A practical solution for scripting language compilers. In *SAC '09*, page 1916–1923. ACM.

Chambers, C. and Ungar, D. (1991). Making pure object-oriented languages practical. In *OOPSLA '91*, page 1–15. ACM.

Chevalier-Boisvert, M. and Feeley, M. (2014). Simple and effective type check removal through lazy basic block versioning. *CoRR*, abs/1411.0352.

Chevalier-Boisvert, M., Hendren, L., and Verbrugge, C. (2010). Optimizing matlab through just-in-time specialization. In *CC '10*, page 46–65. Springer-Verlag.

Cooper, K. D., Hall, M. W., and Kennedy, K. (1993). A methodology for procedure cloning. *Computer Languages*, 19(2):105–117. ICCL '92.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *POPL '82*, page 207–212. ACM.

Dean, J., Chambers, C., and Grove, D. (1995). Selective specialization for object-oriented languages. *SIGPLAN Not.*, 30(6):93–102.

Emami, M., Ghiya, R., and Hendren, L. J. (1994). Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Not.*, 29(6):242–256.

Husák, R., Zavoral, F., and Kofroň, J. (2020). Optimizing transformations of dynamic languages compiled to intermediate representations. In *TASE '20*, pages 145–152.

Míšek, J., Fistein, B., and Zavoral, F. (2016). Inferring common language infrastructure metadata for an ambiguous dynamic language type. In *ICOS '16*, pages 111–116.

Míšek, J. and Zavoral, F. (2010). Mapping of dynamic language constructs into static abstract syntax trees. In *ICIS '10*, pages 625–630.

Míšek, J. and Zavoral, F. (2017). Control flow ambiguous-type inter-procedural semantic analysis for dynamic language compilation. *Procedia Computer Science*, 109:955–962. ANT '17 and SEIT '17.

Míšek, J. and Zavoral, F. (2019). Semantic analysis of ambiguous types in dynamic languages. *Journal of Ambient Intelligence and Humanized Computing*, 10(7):2537–2544.

Ottoni, G. (2018). HHVM JIT: A profile-guided, region-based compiler for PHP and Hack. In *PLDI '18*, page 151–165. ACM.

Rigo, A. and Pedroni, S. (2006). PyPy's approach to virtual machine construction. In *OOPSLA '06*, page 944–953. ACM.

Serrano, M. (2018). Javascript AOT compilation. In *DLS '18*, DLS 2018, page 50–63. ACM.

Tobin-Hochstadt, S. and Felleisen, M. (2010). Logical types for untyped languages. *SIGPLAN Not.*, 45(9):117–128.

Wimmer, C. and Würthinger, T. (2012). Truffle: A self-optimizing runtime system. In *SPLASH '12*, page 13–14. ACM.

Zhao, H., Proctor, I., Yang, M., Qi, X., Williams, M., Gao, Q., Ottoni, G., Paroski, A., MacVicar, S., Evans, J., and Tu, S. (2012). The hiphop compiler for php. *SIGPLAN Not.*, 47(10):575–586.