Optimizing Transformations of Dynamic Languages Compiled to Intermediate Representations

Robert Husák
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
husak@ksi.mff.cuni.cz

Filip Zavoral
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
zavoral@ksi.mff.cuni.cz

Jan Kofroň
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic
jan.kofron@d3s.mff.cuni.cz

***Abstract*—Compiling dynamic languages to stack-based intermediate representations used in platforms such as .NET and Java proved to be useful, mainly due to the enhanced interoperability and security. To produce the best intermediate code possible, current approaches perform a detailed flow-sensitive type analysis of the original code and utilize its results to choose the most efficient operations of the target platform. As known from the traditional compilers, the standard way to further increase program performance is using a set of transformations, which increase its efficiency while preserving its semantics. However, these transformations are not directly usable in the compilers of dynamic languages, because these operate on a higher level of abstraction; moreover, dynamic languages pose specific challenges, such as those stemming from weak typing. In this paper we propose a set of transformations which fit into the architecture of a dynamic language compiler, fitting well together with the type analysis. For evaluation purposes, we implemented them to Peachpie, a compiler of PHP to .NET. Applying the transformations during compilation of WordPress resulted in improvement of 0.6% in the generated assembly size, 1.8% in CPU time and 0.8% in memory consumption.**

***Index Terms*—compilers, optimization, scripting languages, program analysis, program transformation**

I. Introduction

Dynamic programming languages have become very popular in software development, mainly due to their simplicity and transparency. While the common way is to use them in their respective original runtime environments, there have been successful attempts to compile them into a strongly-typed intermediate language used in the platforms such as .NET and Java. Although this approach is not applicable in all cases due to additional deployment complexity, it provides several significant benefits, e.g. enhanced security, interoperability and source-less distribution [1].

One of the challenges such compilers face is to optimize the resulting intermediate language to be as efficient and clean as possible. Although there is a plethora of compiler optimization techniques developed over several decades [2], [3], they are not directly applicable to this kind of compilation and many optimizations aimed at dynamic languages [4], [5] are not available at compile time. The specific difficulties are explained in section II.

This work was supported by Charles University Grant Agency (GA UK) project 896120, the project PROGRESS Q48 and the grant SVV-2017-260451.

This paper describes a way to extend a dynamic-language compiler with an optional optimization phase, which performs semantics-preserving transformations. The work is based on Peachpie [1], a compiler of PHP into .NET. Therefore, the examples of particular transformations are bound to the specifics of PHP, but the general idea is transferable to other languages and platforms as well. Our contributions are as follows:

- We describe the overall compilation algorithm with an optional optimization phase in section III.
- Section IV explains the optimizing transformations based on the observations of certain usage patterns which can be simplified.
- Two transformations requiring more sophisticated analysis are shown in section V.
- We evaluate the overall effects and overhead of the transformations in section VI.

In section VII, we compare our approach to the most relevant existing work, while section VIII concludes.

II. Problem Statement

A traditional way to create a compiler is to implement the whole process from parsing the source code to its execution. Although this approach requires the most effort, its main benefit is the opportunity to make all the decisions regarding the infrastructure of the compilation/execution workflow. When compiling an existing language to an existing platform, we find ourselves in a completely opposite situation. The expected behavior is already specified and the underlying architecture is already implemented; therefore, we have to tailor the architecture of the respective compiler to the needs and best practices of both.

A managed platform such as .NET provides us with a potentially very efficient runtime environment containing just-in-time (JIT) compilation. A JIT compiler accepts a high level stack-based intermediate language and compiles it into the machine code. During the JIT compilation, it usually transforms the code to a reasonably alterable type of intermediate representation (IR) and performs various well-known optimizations such as routine inlining, loop invariant code hoisting, copy propagation and common subexpressions elimination [6]. However, these optimizations, as well as IR,

978-1-7281-4086-5/20/$31.00 ©2020 IEEE
DOI 10.1109/TASE49443.2020.00028

are internal parts of the respective platform and are applied at runtime. Therefore, we cannot utilize them directly when compiling a program to the code in a stack-based intermediate language. Instead, the optimizations we target in this paper must be performed on a higher level of abstraction so that the input for the JIT compiler is as simple as possible.

Stack-based intermediate languages naturally correspond to recursively traversing a syntax tree and emitting its contents in post-order. For example, the addition of two integers `a + b` can be represented as the tree `Add(Var(a), Var(b))` and emitting its contents to a toy stack-based intermediate language yields `loadVar a; loadVar b; add`.

A natural approach to optimization present in Peachpie is to analyze the code and annotate the tree with pieces of information potentially usable to produce better code in the emit phase [7]. As an example, consider the following PHP code:

```
function foo(array $a, array $b) {
  $b["b"] = (isset($a["a"]) ? $a["a"] : "b");
  return $b;
}
```

If we compile it using Peachpie to the Common Intermediate Language (CIL) and decompile it to C#, the result looks similar to this:

```
PhpValue foo(PhpArray a, PhpArray b)
{
  a = a.Copy();
  b = b.Copy();
  PhpValue v =
    Operators.OffsetExists(a, "a")
    ? a.GetItemValue("a")
    : "b";
  b.SetItemValue("b", v.GetValue());

  return b.Copy();
}
```

Notice the return type `PhpValue`, which is a structure possibly containing a PHP value of any type, e.g., a string, an associative array or an object reference. Whenever we cannot infer the exact type of some expression by data-flow analysis, `PhpValue` is used to defer the particular operation selection to runtime [1]. On the other hand, both `$a` and `$b` must contain associative arrays (`PhpArray` class in Peachpie); therefore, we can emit strongly-typed static calls to the `Copy`, `OffsetExists`, `GetItemValue` and `SetItemValue` methods.

The first two calls to `PhpArray.Copy` are there to mimic passing the function parameters by value[1]. Next, the result of the conditional expression is stored to a temporary variable `v`. Notice that before being added to `b`, `GetValue` is called on `v`. The reason is that one of the possible types `PhpValue` can be is the class `PhpAlias`, used to implement reference assignments [8]. As an alias might have been possibly stored in `$a["a"]`, we must explicitly dereference it using `GetValue`. The final call to `Copy` is there to mimic the return by value.

In this work, we introduce a set of optimizing transformations, which can make a code like this even more efficient. The current code in the conditional expression requires two consequent lookups to the associative array using the same key. If we recognize this access pattern and replace it with a simple dedicated operation `TryGetItem`, we can simplify the access as follows:

```
PhpValue v =
  a.TryGetItemValue("a", out var w) ? w : "b";
```

Another way to improve the resulting code is to erase the call to `Copy` in `return`. We can afford to do that without changing code semantics, because `b` was already copied at the beginning of the function.

By enabling transformations of this kind, we can possibly gain numerous benefits in terms of the performance and the size of the generated code. On the other hand, it introduces additional complexity to the compiler. For example, if we decide to remove unreachable parts of code, it gives us the opportunity to re-run the type analysis and make it more precise. Therefore, the architecture of a dynamic language compiler must be updated to enable these transformations. Our approach in this direction is described in the following section.

III. Architecture

The purpose of this section is to explain where in the compilation process the optimizing transformations should take place. The overall compiler architecture and its internal structures are inspired by Peachpie, but it should be feasible to adapt them to other compilers as well. Notice also that the novelty lies mainly in the application to dynamic language compilers, since these ideas have been present in traditional compilers for a long time [2], [3], [9].

The suggested way to accommodate optimizing transformations into a dynamic language compiler is shown in Fig. 1. At first, let us recapitulate the phases already known from previous work on Peachpie. `ParseSyntaxTrees` transforms source files into a set of abstract syntax trees (AST). ASTs are then processed and the structure of particular routines is then transformed into control flow graphs (CFG). It is important to note that although the control flow is captured using edges between particular nodes, each expression is still contained in a tree structure, being called a semantic tree [7], [10]. The nodes in semantic trees contain annotations important for their later `Emit` to an intermediate language, such as their access mode. Further information is then obtained by `Analyse` using a flow-sensitive interprocedural analysis to determine, among others, possible types of all expressions and constant values known during compile time.

The contribution of this paper is in the `Transform` phase. Because it modifies the CFGs, we need to re-run `Analysis` each time it is successful. More precise analysis results yield more opportunities to transform the code; therefore, we run the analysis and transformation phase in a cycle. To limit its

[1]Note that the proper copy-on-write semantics is implemented in `PhpArray`, but we consider it as a hard copy for the sake of simplicity.

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

Fig. 1. High-level overview of a dynamic language compilation algorithm extended with optimizing transformation phase.

```
Transform(CFGs)
  for all cfg ∈ CFGs do
    TransformNode(cfg.enter)

TransformNode(n)
  if n.visited then
    return
  else
    n.visited ← true

  for all op ∈ n.operations do
    RewriteOperation(op)

  if n.next is [n1, n2] and n.condOp is const then
    n.next ← n.condOp ? [n1] : [n2]

  for all n_next ∈ n.next do
    TransformNode(n_next)

RewriteOperation(op)
  for all child ∈ op.children do
    RewriteOperation(child)
  op ← TransformOperation(op)
```

Fig. 2. The implementation of the `Transform` phase.

impact to compilation time, the number of maximum iterations can be limited by `maxIters`, e.g. set to zero in debug mode.

`Transform` works by traversing and simultaneously modifying CFG, as shown in Fig. 2. Starting from the entry node, it explores the CFG using a depth-first search, marking the already visited nodes. The first optimization comes at this point. If a node `n` targets two nodes `n1` and `n2` on a condition whose value is known to be constant, it removes the unreachable target from the CFG and processes only the valid one. Currently, it is the only modification performed on the CFG structure, but we plan to add more of them in the future.

As all operations performed on a CFG node are stored as semantic trees, a natural way to modify them is using recursion. In our algorithm description in Fig. 2, `RewriteOperation` is responsible for traversing the tree, while `TransformOperation` contains the main logic for all the transformations described in the rest of this paper.

A sample PHP code with opportunities for transformation is shown in Fig. 3. The first pass of `Analyse` discovers that

```
function bar() {
  if (function_exists("print_r")) {
    $b = 42;
  } else {
    $b = "bar";
  }

  if (is_string($b)) {
    return "bar";
  } else {
    return 24;
  }
}
```

Fig. 3. A sample PHP code which requires multiple phases of analysis and transformation to reach the optimal result.

`$b` can be either a string or an integer and the expression `function_exists("print_r")` is always true, because it is in the referenced core library. Due to the constant value in the `if` clause, `Transform` replaces the whole first `if..else` construct by `$b = 42`. During the next analysis phase, `$b` is proved to possibly be only integer and `is_string($b)` to be false. Again, the transformation phase removes the second `if..else` construct, leaving only `return 24`. The final analysis phase infers that the return type of `bar` is integer.

IV. Pattern-based Transformations

As explained in the previous section, expression transformations are performed by recursively traversing and updating the semantic trees. To determine whether to transform a particular node, it is matched against a set of patterns and their additional conditions. If the match succeeds, the transformation is performed by replacing the node with a newly created one.

Table I shows the most interesting transformations performed in our work, sorted by their increasing complexity and numbered in the first column. The second column contains an intuitive idea behind each transformation, explained using code snippets in a mixture of PHP and C# syntax. A single-letter identifier with `$` as a prefix stands for an arbitrary variable, whereas a single-letter identifier without a prefix stands for an arbitrary subexpression. The third column presents the precise pattern which an expression subtree needs to match in order to be transformed. It is expressed in a prefix notation where node type names and enumeration values start with upper-case letters. Lower-case identifiers represent arbitrary subexpressions, which can be captured to be used in the resulting subtree. Some transformations express additional conditions on the captured subexpressions, possibly using information from the analysis phase. The fourth column shows the structure of the resulting subtree.

Let us demonstrate the notation on the first transformation, turning a double logical negation into a single conversion to Boolean. The identifier `x` symbolizes an arbitrary expression nested inside two logical negations. `UnaryOp(operator, operand)` is a semantic tree node representing an application

TABLE I
PATTERN-BASED TRANSFORMATIONS

No.	Intuition	Pattern and Conditions	Result
1	`!!x` → `(bool)x`	`UnaryOp(Negate, UnaryOp(Negate, x))`	`Conversion(Bool, x)`
2	`x * -1` → `-x`	`BinaryOp(Multiply, x, Literal(-1))`	`UnaryOp(Minus, x)`
3	`empty($x)` → `true`	`Empty(Var(x))` *type*(`x`) = {} ∨ *type*(`x`) = {`null`}	`Literal(True)`

4	`dirname(__FILE__)`
	↓
	`__DIR__`
		`FunctionCall(`
		`  \dirname,`
		`  PseudoConst(File))`
			`PseudoConst(Dir)`

5	`ord(s[i])`
	↓
	`GetItemOrdValue(s, i)`
		`FunctionCall(`
		`  \ord,`
		`  ArrayItem(s, i))`
		type(`i`) = {`integer`}
			`ArrayItemOrd(s, i)`

6	`isset($a[i]) ? $a[i] : x`
	↓
	`TryGetItem($a, i, out v)`
	`? v : x`
		`Conditional(`
		`  IsSet(ArrayItem(Var(a), i)),`
		`  ArrayItem(Var(a), i),`
		`  x)`
		`i = Var(*)` ∨ `i = Literal(*)`
			`ArrayItemTry(Var(a), i, x)`

7	`(callable)"fn"` → `fn`	`Conversion(Callable, Literal(fn))` *routine*(`fn`)	`CallableConvert(fn)`

8	`(callable)["class", "m"]`
	↓
	`class.m`
		`Conversion(`
		`  Callable,`
		`  Array(Literal(class), Literal(m)))`
		routine(`class::method`)
			`CallableConvert(class::m)`

9	`(callable)[$this, "m"]`
	↓
	`this.m`
		`Conversion(`
		`  Callable,`
		`  Array(Var(this), Literal(m)))`
		type(`this`) = {`class`} ∧ *routine*(`class::m`)
			`CallableConvert(class::m)`

of a unary `operator` on an `operand`. In this case, we are interested in the `Negate` operator. To represent the double application, another `UnaryOp` is used as the `operand` of the first one. The nested expression `x` is captured from the original subtree and used in its replacement by the conversion to Boolean, represented by `Conversion(Bool, x)`.

The second transformation is a subtle simplification as well, replacing a multiplication by -1 to a direct application of unary minus. Although the JIT compiler itself can perform this optimization, it is applicable only if `x` is of a single numeric type. Even though the type analysis can prove it to be, it is not a common situation, e.g. due to an implicit cast of overflowing integers to floating-point numbers in PHP. Our transformation can simplify the code independently of the type.

The third transformation uses the previous analysis phase to evaluate the `empty($x)` expression. Notice that a variable reference is expressed in the semantic tree as `Var(v)` node, where `v` is the unique identifier of a particular variable. The function *type* is created during the previous analysis and associates each expression and variable with a set of its possible types. If the set is empty or contains only `null`, the variable was either never assigned to or was unset by being assigned `null`. Therefore, in this case, we can safely replace the original expression by `true`.

The popularity of the construct `dirname(__FILE__)` is caused, among other things, by its backward compatibility with PHP 5. Because it is semantically equivalent to `__DIR__`, we can safely replace it using the fourth transformation. Apart from the removed function call, it helps to statically resolve file inclusions. Notice that we use a special node `PseudoConst` to denote this type of constants and `FunctionCall(f, arg1, arg2,...)` to call the function `f` with the given arguments. To denote that the function being called must be the one from the global namespace and not a function with the same name from a local namespace, we identify it with the `\` prefix in the pattern.

A string in PHP has an interesting behavior when one of its characters is read using an indexer. Instead of directly returning its ASCII value, a single-character string is created and retrieved. Therefore, a common way to access the ASCII value of a given character is to apply the indexer and then use the function `ord`, which retrieves the ASCII code of the first character of the obtained string. This approach, however, leads to unnecessary repeated allocations of single-character strings and the immediate retrieval of their values. Since we are able to identify this pattern, we can replace it with the

custom operation `GetItemOrdValue` in the fifth transformation. Although we require the index `i` to be integer, `s` can potentially be of any type. However, `GetItemOrdValue` has a special overload with string as the first parameter and the custom node `ArrayItemOrd` may choose it during the emit phase if *type*(`s`) = {`string`}. The string overloading works simply by retrieving the character on the given position. The general version of `GetItemOrdValue` checks the type in runtime and when it is not a string, it mimics the behavior of the original expression.

Another commonly used expression is shown in the sixth transformation: determining whether an array element exists using `isset` before its retrieval. We can reduce the two array lookups to one by combining the semantics into the custom operation `TryGetItem`. Again, there are several overloads based on the argument types, from which the appropriate one is selected during the emit phase of the custom node `ArrayItemTry`. Notice that we constrain the index `i` to be either a variable reference or a literal to ensure that it does not cause any side effects.

The transformations 7, 8 and 9 revolve around the logic of callbacks in PHP. There are several library functions accepting a callback as an argument, e.g. `call_user_func` or `array_map`. Aside from using closures, PHP allows the caller to pass a string or a two-element array to identify the defined method. It is then converted at runtime to the appropriate delegate by dynamically finding the corresponding routine using reflection. Our transformations aim to move this lookup to the compilation phase. The first transformation captures the situation when a single string is being converted to a callback, meaning that the global function of that name should be used. Notice that the pattern requires the string value to be known during the compilation. Furthermore, the function *routine* is used to determine whether a routine with the given name exists. If the transformation is applied, the resulting expression `CallableConvert` is responsible for emitting a lazily-loaded delegate. The similar situation is in the second transformation, only in this case, the two-element array specifies a class and its static method. As we can see in the last transformation, an instance method can be resolved during compilation as well, supposing it is called on `this`. Although the resulting delegate still needs to be recreated upon each call, we save CPU time and memory allocation by skipping the creation of the array and the lookup of the method using reflection.

The transformation list is not exhaustive; there are several other transformations similar to the first four, which we skipped due to space restrictions. Examples include evaluation of operations whose operands are known at compile time, such as constant string concatenation.

Also, there is room for adding more transformations, possibly by drawing inspiration from existing dynamic language runtimes and traditional compilers. Variants of the aforementioned transformations can already exist in such contexts. The transformations described in the next section more aim at the specific issues of dynamic language compilation, so they can be classified as more original.

```
function foo($a, $b) {
  bar($a);
  if (is_array($b)) {
    $c = $b;
  } else {
    $c = [];
  }
  return $c;
}
```

Fig. 4. A sample PHP code containing several places where values are copied.

```
PhpValue foo(PhpValue a, PhpValue b)
{
  a.PassValue();
  b.PassValue();
  bar(a);
  PhpValue c;
  if (Variables.is_array(b)) {
    c = b.Copy();
  } else {
    c = (PhpValue)PhpArray.NewEmpty();
  }
  return c.Copy();
}
```

Fig. 5. The PHP sample of value copying compiled by Peachpie to CIL and decompiled to C#.

V. Data-flow-based Transformations

The transformations shown in section IV are based on discovering certain expression patterns in semantic trees and simplifying them. Some patterns use information from the previous type-analysis phase as well as the knowledge of existing classes and routines. However, none of these transformations has to perform a custom flow-sensitive analysis. This section presents a set of transformations which need this kind of analysis in order to be applied. These transformations aim at reducing the number of calls to helper methods which maintain the PHP assignment-by-value semantics. Consider the PHP code example in Fig. 4 and the result of its compilation in Fig. 5.

As mentioned in section II, `PhpValue` is a helper structure which can possibly hold a value of any valid PHP type or an instance of the class `PhpAlias`. Therefore, just simply passing a `PhpValue` argument to a function in the way common in .NET can cause unwanted aliasing to be introduced. To prevent that, `PhpValue` contains the method `PassValue`, which ensures that any referenced object is properly extracted from `PhpAlias` and copied if necessary. By default, it is emitted for each argument. Also notice the two calls to `Copy`: the first one in the assignment `$c = $b` and the second one in the return statement.

Most of the mentioned calls are not necessary. The parameter `a` is only passed to `bar` and never used anywhere else. The function `bar` expects that its argument can be an alias

TABLE II
DATA-FLOW-BASED TRANSFORMATIONS

No.	Intuition	Pattern and Conditions	Result
1	`p.PassValue() → ;`	`PassParam(Var(p))` $\forall$e $\in$ *Exprs* (e = `Var(p)` $\Rightarrow$ *safeUse*(e) $\wedge$ $\neg$*afterCall*(e))	`NoOp`
2	`x = y.Copy()` → `x = y`	`Assignment(Var(x), c)` `c = Copy(Var(y))` $\wedge$ $\forall$e $\in$ *Exprs* (e = `Var(v)` $\wedge$ *reaches*(c, e) $\Rightarrow$ $\neg$*modified*(e))	`Assignment(Var(x), Var(y))`
3	`return x.Copy()` → `return x`	`Return(Copy(Var(x)))` $\neg$*ref*(x)	`Return(Var(x))`

or a reference to `PhpArray` and can handle these situations itself. As a result, we can remove `a.PassValue`. On the other hand, we cannot do the same for `b` due to two reasons. First, when being assigned to `c`, it is not expected to hold an alias, so we would need an additional call to `GetValue` there. Second, if `b` is an alias, `bar` can modify its value, changing the resulting semantics. Regarding `Copy`, it is possible to eliminate all its calls in this case, as `c` is never modified after being assigned the value from `b` and `b.PassValue` makes sure that returning `c` does not cause aliasing with `b`.

The corresponding transformations are shown in Table II. Its structure is the same as in section IV, only the second column is in C# syntax and the pattern conditions are more complex and depend on information not directly available from the analysis phase. As a result, these transformations can be implemented in the same way as the simpler ones, assuming that the additional analysis is performed beforehand. As the transformation phase is optional and this analysis is only useful for the transformations, there is no need to perform it in the main analysis phase.

The removal of `PassValue` in the first transformation is expressed as replacing the node `PassParam` by an empty operation. *Exprs* in the condition stands for the set of all the semantic tree nodes in the given routine. By enumerating it with `e` matched against `Var(p)`, we inspect all the references of the parameter `p` in the routine. There are two predicates supplied by the analysis: *safeUse* and *afterCall*. The purpose of *safeUse* is to determine whether `p` is accessed in a context which does expect it to possibly be a hidden alias or a reference to an existing object outside the routine scope, such as an array or a string. Its conservative implementation is to return *true* if and only if the parameter is passed by value to a routine call. The predicate *afterCall* marks the regions of the program occurring after a possible call to an external routine. Any aliases not dereferenced before such call can be subject to modification during it, so it is not safe to use them afterwards. Notice that apart from directly calling a routine, methods such as `__clone`, `__get`, `__toString` or `offsetExists` can be indirectly called by cloning, field value retrieval, implicit conversion to string etc. Therefore, all the operations which can cause these calls should make *afterCall* return *true* as well. Computing *afterCall* is straightforward, as it is basically a reachability problem solvable by coloring the operations in CFG.

The second transformation presents the removal of copies in assignments, which requires two other predicates: *modified* and *reaches*. The former one captures whether the given variable reference in its current context can be used to modify the contents of the underlying structure. For example, in the expression `$x["a"] = 42` variable `$x` can be modified, so if the last assignment to `$x` was `$x = $y`, its copy operation must be kept to prevent the modification of `$y`. The predicate *reaches* connects each `Copy` expression in an assignment with all the uses of variables whose aliasing it prevents. Basically, the `Copy` in `x = y.Copy()` reaches all following occurrences of both `x` and `y` until any of them takes part in another assignment. If there is assignment `z = w` whose copy was removed in a previous transformation round, all the copies that reached `w` now reach `z` as well. Note that in Peachpie, we use a fixpoint analysis [11] based on bit mask states to compute *reaches* efficiently (the details are beyond the scope of this work).

Interestingly, removing `Copy` from variables in return statements is much simpler than the previous transformations. The reason is that the validity of all the local variables ends with the routine, so we do not have to care about their possible aliasing anymore. Note also that even if we directly or indirectly return a parameter, its usage in an assignment or a return statement prevents `PassParam` from being removed. The only situation when we cannot eliminate a copy upon return is when the variable can be a reference. We can check this situation using *ref* from the main analysis phase.

VI. EVALUATION

There are three important research questions when it comes to evaluating the aforementioned transformations:

- **RQ1:** *What are the measurable gains of each transformation?*
- **RQ2:** *Are the transformations applicable in real-life projects?*
- **RQ3:** *Are there any measurable benefits in applying the whole set of transformations to real-life projects?*

In order to answer them, the transformations were implemented in Peachpie. Using the library BenchmarkDotNet [12],

TABLE III
EVALUATION OF TRANSFORMATION SAVINGS AND APPLICABILITY

Transformation	Savings		Occurrences in WordPress
	Time [ns]	Memory [b]	
DoubleNegate	1	0	0
MinusOneMultiply	2	0	6
EmptyRemoval	20	0	0
DirnameSimplify	426	488	164
OrdString	404	72	69
TryGetItem	52	0	373
CallableFunction	58,544	609	275
CallableStatic	1,486	856	13
CallableThis	1,233	744	37
ParamCopyRemoval	41	32	1,726
AssignCopyRemoval	58	32	243
ReturnCopyRemoval	50	32	2,423

TABLE IV
EVALUATION OF TRANSFORMATIONS ON WORDPRESS

Metric	Original	Transformed	Difference
Transformation cycles	0	6	
Compilation time [s]	47	61	+30%
Assembly size [kiB]	16,155	16,051	-0.6%
Request time [ms]	103.9	102	-1.8%
Request memory [MiB]	13.22	13.11	-0.8%

we developed two sets of benchmarks. The first one uses short microbenchmarks, where each one is tailored for a particular transformation. Its purpose is to measure the gains of each transformation separately. The second set of benchmarks is a simulation of sequential requests to the main page of a website implemented in WordPress 3.5.1. The experiments were conducted on a desktop with an Intel Core 2 Quad Q9300 2.5GHz CPU and 6GB RAM. Apart from runtime benchmarks, compiling existing applications provided us with information about the number of performed transformations, compilation time and the resulting assembly size.

Table III presents the results in terms of savings and applicability of each transformation. The first two columns contain the savings provided by a single application of the given transformation in the microbenchmarks. They are computed by compiling the given code snippet with and without transformations allowed and comparing the time and memory usage differences over multiple runs. The last column shows the number of applications of the given transformation when compiling WordPress.

The impact on the selected projects is shown in Table IV. We observe a set of metrics, comparing the original version where the transformations are disabled with the one where they are enabled. Request time and total allocated memory are means across multiple repeated requests to the front pages of the respective applications.

Let us answer the research questions with the support of these data:

RQ1: As we can see in Table III, each transformation performed separately has a positive impact on the runtime resources and assembly size. The highest savings are provided by compile-time `callable` conversions, whereas the lowest savings are achieved by manipulations of simple expressions, such as the transformation of `x * -1` to `-x`. Note that the usual savings of CPU time are in tens of nanoseconds and the usual savings of allocated memory are in tens of bytes. Therefore, in order for these transformations to make any measurable effect in larger projects, they must be applied sufficient number of times, especially in code which is executed often. This brings us to the second research question.

RQ2: As seen in Table III, most transformations are performed reasonable number of times in real projects, giving them chance to decrease the assembly size, increase the precision of the analysis and save runtime resources.

RQ3: Table IV shows that the effects of transformations are visible even on large projects. Understandably, the impact is not as radical as it might be in specialized microbenchmarks. However, it shows that the transformation phase is worth the reasonably increased compilation time and provides opportunities for future improvements.

VII. RELATED WORK

This paper extends the approach to PHP analysis and compilation in the way it is performed in Phalanger [13] and Peachpie. The addition of transformation phase utilizes the information gathered during the previous analysis and provides a way to make the subsequent analysis more precise.

Since PHP 7.1, a static optimization phase of its bytecode [14] has been added to Zend Engine [15], the original PHP interpreter. The bytecode is optimized by being transformed to static single assignment (SSA) form, upon which customized versions of traditional optimizations are performed, e.g. constant propagation and dead code elimination. Whereas there are several ideas and insight which might be useful for our approach, PHP bytecode and Zend Engine are vastly different from CIL and .NET. For example, Zend Engine does not need a special way to handle aliases, but it must ensure the proper reference counting of all the objects. Therefore, the optimizations often target different aspects of the execution.

There are several other alternatives to Zend Engine. The HipHop Virtual Machine (HHVM) [16] developed by Facebook was originally a complete runtime environment with dynamically optimized JIT compilation customized for PHP. Recently, it dropped general support for PHP in favor of Hack. The main difference comparing to our approach is that HHVM performs most of the optimizations at runtime, whereas we perform them statically when compiling PHP to CIL. Quercus [17] and JPHP [18] compile PHP code to Java, HippyVM [19] uses RPython. We were unable to find evidence of any particular compilation optimizations they perform in their respective documentations. Instead, most mentioned optimizations target the efficiency of used data structures, e.g. associative arrays.

VIII. Conclusion

In this work, we have shown how to implement high-level optimizing transformations in a dynamic language compiler. The main idea is to add a transformation phase to the compilation workflow, which is repeatedly run after the analysis. The transformation of the expression semantic trees is performed by matching their subtrees with a set of patterns and replacing them by different ones on success. When a more sophisticated analysis is needed, it is performed prior to the transformation itself and its results can be then used in the aforementioned patterns.

The suggested approach together with a set of relevant transformations were implemented to the working version of Peachpie. The results have shown that these transformations can reduce the size of generated assemblies and decrease resource consumption at runtime.

The impact on large projects is currently not highly significant, but our work enables for more advanced analyses and transformations that can be employed in the future. As to our future work, we want to explore the possibilities of type specialization on the level of code blocks, routines or routine groups to decrease the amount of type checks. Another direction is optimization of associative arrays, e.g. their replacement by specialized data structures where possible. These techniques require more advanced interprocedural analysis and smart heuristics to prevent them from hampering the performance instead of improving it.

Acknowledgment

We would like to thank Jakub Míšek for his advice regarding several edge cases of PHP semantics and their handling in Peachpie. The mentioned transformations were developed in collaboration with him.

References

[1] J. Míšek and F. Zavoral, "Semantic analysis of ambiguous types in dynamic languages," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 7, pp. 2537–2544, Jul 2019. [Online]. Available: https://doi.org/10.1007/s12652-018-0731-5

[2] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[3] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[4] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, and et al., "Trace-based just-in-time type specialization for dynamic languages," *SIGPLAN Not.*, vol. 44, no. 6, p. 465–478, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542528

[5] C. Wimmer and T. Würthinger, "Truffle: A self-optimizing runtime system," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 13–14. [Online]. Available: https://doi.org/10.1145/2384716.2384723

[6] A. Krall, "Efficient javavm just-in-time compilation," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, Oct 1998, pp. 205–212.

[7] J. Míšek and F. Zavoral, "Mapping of dynamic language constructs into static abstract syntax trees," in *2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, Aug 2010, pp. 625–630.

[8] J. Míšek, B. Fistein, and F. Zavoral, "Inferring common language infrastructure metadata for an ambiguous dynamic language type," in *2016 IEEE Conference on Open Systems (ICOS)*, Oct 2016, pp. 111–116.

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

[10] R. Farrow, "Generating a production compiler from an attribute grammar," *IEEE Software*, vol. 1, no. 4, pp. 77–93, Oct 1984.

[11] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Heidelberg: Springer-Verlag, 1999.

[12] A. Akinshin. (2020, Jan) Benchmarkdotnet. .NET Foundation. [Online]. Available: https://benchmarkdotnet.org

[13] A. Abonyi, D. Balas, M. Beňo, J. Míšek, and F. Zavoral, "Phalanger improvements," *Department of Software Engineering, Charles University in Prague, Technical report*, 2009.

[14] N. Popov, B. Cosenza, B. Juurlink, and D. Stogov, "Static optimization in PHP 7," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 65–75. [Online]. Available: https://doi.org/10.1145/3033019.3033026

[15] (2020, Jan) PHP zend engine. PHP Group. [Online]. Available: http://php.net

[16] G. 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*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 151–165. [Online]. Available: https://doi.org/10.1145/3192366.3192374

[17] (2020, Jan) Quercus: PHP in Java. Caucho Technology. [Online]. Available: http://quercus.caucho.com/quercus-3.1/doc/quercus.xtp

[18] (2020, Jan) JPHP: An alternative to PHP on the JVM. JPHP Group. [Online]. Available: http://jphp.develnext.org

[19] M. Fijałkowski, A. Rigo, R. Gałczyński, R. Lamy, S. Pawluś, A. Oruganti, and E. Barrett. (2020, Jan) HippyVM. Caucho Technology. [Online]. Available: http://hippyvm.baroquesoftware.com