Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications

Sophie Kaleba
S.Kaleba@kent.ac.uk
University of Kent
United Kingdom

Octave Larose
O.Larose@kent.ac.uk
University of Kent
United Kingdom

Richard Jones
R.E.Jones@kent.ac.uk
University of Kent
United Kingdom

Stefan Marr
s.marr@kent.ac.uk
University of Kent
United Kingdom

Abstract

Applications written in dynamic languages are becoming larger and larger and companies increasingly use multi-million line codebases in production. At the same time, dynamic languages rely heavily on dynamic optimizations, particularly those that reduce the overhead of method calls.

In this work, we study the call-site behavior of Ruby benchmarks that are being used to guide the development of upcoming Ruby implementations such as TruffleRuby and YJIT. We study the interaction of call-site lookup caches, method splitting, and elimination of duplicate call-targets.

We find that these optimizations are indeed highly effective on both smaller and large benchmarks, methods and closures alike, and help to open up opportunities for further optimizations such as inlining. However, we show that TruffleRuby's splitting may be applied too aggressively on already-monomorphic call-sites, coming at a run-time cost. We also find three distinct patterns in the evolution of call-site behavior over time, which may help to guide novel optimizations. We believe that our results may support language implementers in optimizing runtime systems for large codebases built in dynamic languages.

CCS Concepts: • **Software and its engineering → Language features**; **Object oriented languages**; **Dynamic compilers**.

Keywords: dynamic languages, call-site analysis, splitting, lookup caches, inlining

DLS '22, December 07, 2022, Auckland, New Zealand

ACM Reference Format:
Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Marr. 2022. Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '22), December 07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3563834.3567538

1 Introduction

Dynamic languages such as JavaScript, PHP, Python, and Ruby are used in industry to build a wide range of systems including application backends. Their dynamic language features support rapid application development, but require run-time compilation and optimization to achieve good performance, because the details needed for optimizations are only known during execution.

Coding practices often encourage small methods and code reuse in various forms. Together with paradigms such as everything is an object [7, 13] or everything is a call [10], method or function calls often end up dominating an application's behavior. Having many calls to many small methods can be expensive if calls are not optimized. Fortunately, widely adopted optimizations such as inline caches [9] minimize lookup overhead by caching the target functions, and inlining, which replaces a function call by the function's implementation in the caller, avoids the call overhead and enables further optimizations.

These optimizations rely on the assumption that the behavior of an application stabilizes over time. In the context of calls, it means that after a while a given call-site is assumed to always invoke the same set of targets. However, other research suggests that applications may undergo distinct phases, each of which may show different behavior [14, 15]. Since dynamic languages are used for increasingly large applications with millions of lines of code, for instance by GitHub and Shopify for their huge Ruby-on-Rails applications [3], or Instagram with millions of lines of Python code,

we are interested in verifying whether these assumptions are still likely to hold.

Language implementations such as TruffleRuby [19], which builds on the Truffle language implementation framework [21] together with the Graal just-in-time compiler [20], combine inline caches with splitting, i.e. the cloning of methods with fresh lookup caches, and optimizations to avoid a duplication of call targets. However, the interplay and effectiveness of these optimizations have not yet been studied in detail.

Ruby is used for many different applications, ranging from personal blogs to large industrial codebases used for instance by AirBnB, GitHub, Shopify, and Stripe. In this paper, we analyze the call-site behavior of a corpus of Ruby benchmarks, which were in part built to guide the optimization of Ruby implementations for and by Shopify [4], the Ruby, and the Ruby-on-Rails communities. These benchmarks include Rails-based applications, document-processing systems, a NES emulator, and others. Specifically, we study the effectiveness of existing optimizations, how they interact, how lookup caches evolve over time and whether they show behavioral patterns, and whether there are differences between the uses of methods and closures. For this work, we instrumented TruffleRuby to collect data, resulting in a corpus of analytical data which we believe will help language developers to identify new opportunities for optimizations.

The key insights described in this paper are:

- Benchmarks categorized by how their lookup caches respond to existing call-site optimizations;
- Industrial Ruby benchmarks tend to be larger and display a higher degree of polymorphism. The majority of calls are monomorphic, as previously reported for other languages in the literature;
- Elimination of target duplicates and splitting are very effective at reducing polymorphism. However, over-splitting is frequent and has additional run-time cost;
- Methods calls are more frequent than closure calls, but there is a slightly higher proportion of closure calls that are polymorphic;
- Run-time call-site behavior can be characterized in three distinct patterns that may help guiding future optimizations.

The main contributions of this paper are:

- A benchmark set consisting of both Ruby benchmarks used for research, and larger industrial Ruby benchmarks.
- A characterization of benchmarks according to their degree of polymorphism to study the relevance and impact of optimizations on call-sites in each category;
- An analysis of the impact of two existing optimizations on method and closure call-site behavior;
- An analysis of the evolution of call-site behavior at run time.

Since starting this work, the TruffleRuby developers have used our feedback to avoid unconditional, and thus overly aggressive splitting of blocks, without harming performance.

2 Terminology and Background

This section gives a brief overview of TruffleRuby, and the terminology and optimization strategies used in modern dynamic language implementations. We also detail the different invokable objects in Ruby, which we will distinguish in our analysis of call-site behavior.

2.1 TruffleRuby, Truffle, and Graal

TruffleRuby is a Ruby implementation based on the Truffle language implementation framework [21] and the Graal just-in-time compiler [20]. TruffleRuby is implemented as an abstract-syntax-tree interpreter, which uses partial evaluation at run time to achieve state-of-the-art performance at the same level as the V8 JavaScript engine on the Are We Fast Yet benchmarks [12]. It is fully compatible with Ruby and able to execute Ruby's native extensions, which allows it to run complex applications, for instance those based on the popular Ruby-on-Rails framework.

2.2 Terminology

In our analysis, we distinguish call-sites and call-targets to characterize the call-site behavior over time.

A **call-site** corresponds to a pair of *(Location, Symbol)*, the symbol being the name of the method called. For instance the code `foo.bar()` on a specific line is the call-site for the method `bar`, applied on receiver `foo`.

The location of a call-site is typically tied to its *lexical location* in the source code. However, methods built into the language runtime do not have a lexical location. Thus, when such methods call other methods, there is no specific code location associated with the call-site. We refer to this kind of location as a *virtual location*, and use the address of the runtime internal representation for this call-site.

A **call-target** is the method actually executed. The same call-site can be associated with different tuples of *(Receiver type, Call-Target)*, the *receiver* being the object on which the method is called (the object `foo` in our previous example).

Finally, we refer to **call-site behavior** to characterize the state of a call-site at run time. In this paper, we use its lookup cache state (see Section 2.3) and content (i.e. the run-time types of its receivers) as a proxy for its behavior. We use this information to characterize a call-site and observe how it evolves at run time.

2.3 Call-Site-related Optimizations

Lookup caches, splitting, and inlining are important optimizations used in dynamic language implementations. TruffleRuby relies on these optimizations for its interpreted and just-in-time compiled performance. While lookup caches

minimize lookup overhead and profile call-site behavior, splitting and inlining reduce the number of call-targets at a call-site to improve performance by reducing the polymorphism. We describe them and briefly detail how they affect call-site behavior.

Lookup caches as described by Hölzle et al. [9] are associated with call-sites and aim to avoid repeated and possibly time-consuming dynamic method lookups. Lookup caches typically hold all call-targets observed at a call-site, up to a certain limit. Thus, instead of a costly traversal of a class hierarchy, possibly performing hash lookups in a method dictionary, the runtime ideally needs to only search linearly through a small array that stores the previously seen receiver types and call-targets.

Lookup caches are usually described as a monomorphic, polymorphic, or megamorphic, depending on how many entries they hold, or more precisely, how many different *(Receiver type, Call-target)* tuples the call-site has observed. Depending on the state of the cache, the implementation strategy used may change. A monomorphic cache contains exactly one entry, which is usually the best case since a method call only needs to confirm that this is the right entry in which case it can be executed immediately. Monomorphism is often also a criterion for inlining, and thus desirable to enable further optimization.

A cache turns polymorphic when it holds more than one entry, up to a certain limit. When this limit is exceeded, the cache is deemed megamorphic, which in most systems means that there is no further caching at this call-site, since too many *(Receiver type, Call-target)* tuples make caching ineffective. The maximum number of entries in a cache is system-dependent. In TruffleRuby, lookup caches turn megamorphic when a ninth entry is added.

Typically, the speed of call resolution depends on the number of entries in the cache, the cache implementation, and the current context. TruffleRuby uses linked lists for lookup caches and new entries are added at the head of the list. In the worst case, the receiver does not match any of the previously seen receiver types, which means that after the full cache traversal, a full method lookup has to be performed.

In this paper, we use the content and state of lookup caches to characterize a call-site's behavior, as it is a starting point for the other call-site optimizations.

Inlining [18] is an optimization that replaces a method call with the body of the call-target. This avoids the call overhead, but perhaps more importantly, in the context of a just-in-time compiler it enables compiler optimizations by enlarging the compilation unit.

Inlining is typically considered if a call-site observes only a single call-target, that is, if the lookup cache is monomorphic.

In TruffleRuby, inlining may happen during JIT compilation. In some cases, TruffleRuby does it already in the interpreter at the AST level, for instance for builtins and specific methods of the core Ruby library (see Section 2.4).

Splitting [9], also called method cloning, duplicates a method in the context of the caller. The new copy starts with empty lookup caches to monomorphize potentially polymorphic call-sites. It is similar to inlining, but does not embed the code and thus, keeps the two compilation units separate.

In TruffleRuby, splitting is done at the method level when a method is called. It copies an uninitialized version of the target method for use at that call-site. This means a call-site has its own copy of the call-target, and all call-sites inside the new copy have uninitialized lookup caches. Ideally, splitting prevents these caches from becoming polymorphic.

During a method call, TruffleRuby, or more specifically the Truffle framework, checks whether a method contains, e.g., polymorphic call-sites. It also checks a flag for splitting requests: this is used to propagate splitting up the caller chain. When a call-target has several call-sites, e.g. a standard library method that is called in many places, the source of polymorphism is assumed to come from the different call-sites and only the called method is split. If there is only a single call-site for a call-target, the polymorphism is assumed to come from higher up in the call chain, and the caller method is marked for splitting.

In the example in Figure 1,[1] `capitalize()` on line 2 becomes polymorphic after being indirectly called on a Symbol and String from lines 10 and 11. In this case, splitting is propagated up the call chain until either a call-target linked to several call-sites is encountered or a propagation limit is reached (five callers in Truffle). In this example, `kapitalize` is marked for splitting so that the calls on lines 10 and 11 will get separate copies.

Additionally, TruffleRuby will force splitting for some known methods in the core Ruby library, so that they specialize in the context where they are used (see Section 2.5.1).

2.4 Methods and Closures in Ruby

In Ruby, similarly to other languages, closures are frequently used in addition to normal methods. In the context of understanding call-site behavior, it is useful to distinguish the two since they may be used differently by programmers.

Methods are defined on a class and, when used, have a receiver that may be lexically explicit (as in `foo.bar()`) or implicit (as in `puts "foo"`, where the receiver of `puts` is the top-level `main` object). While most methods are defined in Ruby code, some are *builtin* and implemented in the Ruby implementation. This includes some methods of Ruby's core library. In TruffleRuby, these builtins are implemented in Java and include for instance methods on `Array` and `TrueClass`.

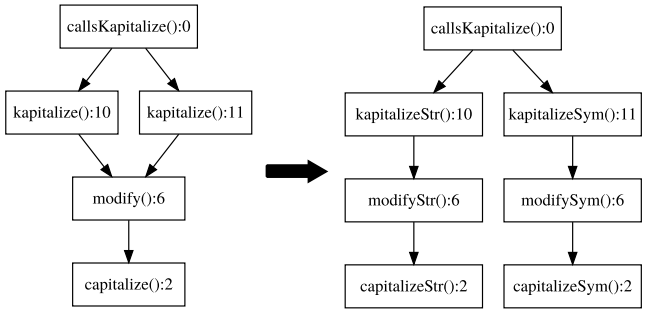
[1] Adapted from https://github.com/oracle/graal/blob/master/truffle/docs/splitting/Splitting.md

```ruby
def modify(arg1)
    arg1.capitalize()
end

def kapitalize(arg1)
    modify(arg1)
end

def callsKapitalize()
    kapitalize("foo") # a String
    kapitalize(:bar)  # a Symbol
end
```

(a) Snippet of code triggering splitting



(b) Impact of splitting on the application's structure

Figure 1. Code example that requires recursive splitting, i.e. splitting is propagated up the call chain so that the resulting call tree gets two distinct, specialized branches: one for `Symbol` and one for `String`.

TruffleRuby will inline many of these builtins at the AST-level to avoid the call overhead, and allow specialization in the context of the caller method.

Closures are called blocks and exist in two flavours in Ruby, as *procs* or *lambdas*. Lambdas are similar to standard methods but check their arguments more strictly. Procs are more forgiving of mismatching arguments. Block literals in the code instantiate as procs. In this paper, we will not distinguish these flavours further.

2.5 Call Optimization in TruffleRuby

In addition to lookup caches, inlining, and splitting, TruffleRuby uses further optimizations that affect method calls.

2.5.1 Eliminating Call Target Duplication Classic lookup caches contain entries of *(Receiver type, call-target)* (see Section 2.3). In Truffle, method calls are realized with the framework's call node (called `DirectCallNode`), which takes the call-target and implements splitting and triggering of the just-in-time compiler. So, in Truffle languages, a lookup cache would contain a *(Receiver type, call node)* tuple.

This approach, however, does not account for code reuse in inheritance hierarchies and Ruby features such as mixins or class patching. With these features being widely used, it is common that lookups on different types of receivers resolve to the same call target, such as in this snippet of Ruby code:

```ruby
list_of_ints = [68719476736, 2]

for e in list_of_ints do
    e.ord()
end
```

The two elements of `list_of_ints` are both integers, but the first is an instance of the class `BigNum` and the second a `FixNum`. They share the same `ord` implementation, i.e. the same target. This call-site is polymorphic if we consider only the receiver's class, but it is monomorphic when considering the target.

TruffleRuby solves this by introducing a two-level caching strategy. The first level is equivalent to the classic cache and contains *(Receiver type, call-target)* tuples. The second level is used to eliminate duplicate call-targets and contains *(call-target, call node)* tuples. This means that there is only a single call node per call-target, instead of possibly multiple call nodes for the same call-target, as in the classic design. This de-duplication is beneficial, for instance to enable inlining.

2.5.2 Additional Splitting in TruffleRuby TruffleRuby adds further splitting strategies on top of the ones in Truffle.

By default, closure application sites split the closure's call target.[2] Thus, all entries in the lookup cache at an application site contain call targets that can fully specialize in the context of the caller.

Some methods in Ruby's core library are treated similarly and are always split, for instance `Array.insert` and `String.tr`. For some other methods, splitting is disabled, typically to manage the cost/benefit of splitting.

3 Methodology

This section describes the benchmarks which we use to conduct our call-site behavior analysis and the methodology used to gather the data for our analysis.

3.1 Selection of the Benchmark Set

We use benchmarks from the Are We Fast Yet project [12], which contains 9 microbenchmarks and 5 slightly larger ones that were designed for comparing performance between different programming languages. We use these benchmarks because they are small, well understood, and allow us to verify the results of our analysis.

In addition, we use benchmarks from the TruffleRuby and YJIT projects [4]. These are larger application-level benchmarks collected to guide the development of optimizations for the two corresponding Ruby implementations. YJIT's benchmarks are of particular interest since they are meant to resemble the real-world workloads of Shopify.[3]

[2] After discussing our results with the TruffleRuby team, this has been changed, and closure calls rely on Truffle's normal splitting.
[3] Shopify funds YJIT's development.

These benchmarks include web server applications such as ERubiRails and BlogsRails, which are built on top of the large Ruby-on-Rails framework and rely on template processing. Since Ruby is often used for such web applications, they represent a major use case and optimization target. In addition, we use OptCarrot (a NES emulator), AsciiDoctor (a text processing system), HexaPDF (a PDF render), the Jekyll static site generator, and various other benchmarks built from typical parts of large e-commerce applications.

3.2 Behavior Monitoring and Analysis

We use an instrumented version of TruffleRuby to capture the data for our call-site analysis. This instrumentation logs both methods and closure activations, and is triggered on each call. Exceptions are detailed in Section 4.7.

Trace description. Figures 2–4 show excerpts of an execution trace for the DeltaBlue benchmark. Each line of the trace represents one call. The first number is the number of the line in the trace, for instance `493` on the first line of Figure 2. The second element is the lexical location of the call (`./deltablue.rb:47` is the file name and line number). The third element is the symbol name (`execute`), the fourth (`EditConstraint`) and fifth (`EditConstraint`) are the receiver type and the type in which the call-target is actually defined. We use this distinction to differentiate between receiver and target types (see 2.5 for a definition, and Figure 3 for an example from a trace). The sixth item `1397` is an ID that uniquely identifies the virtual call-site: we use this information to monitor whether splitting occurred for the call-site. Lastly, the last item `1894` represents the lexical call-site ID. This ID is required because the lexical location fetched by the parser operates on the granularity of a line. This means that, without this ID, our instrumentation would have treated two call-sites present in the same line with the same symbol as one call-site being called twice.

We use R to analyze these traces, compute statistics, and generate summary plots and tables.[4] The results of the analysis are described in detail in Section 4.

Behavior reconstruction. We use the trace to assess the state of the call-sites' behaviors and to identify the impact of call-site optimizations. The lookup cache state is reconstructed by counting the number of different receivers at a given call-site. For example, in Figure 2 the call-site of `execute`, located at `./deltablue.rb:47`, has been called three times, with two different receivers, `EditConstraint` and `EqConstraint`: we conclude this call-site is polymorphic, and that its lookup cache has two entries.

We identify the potential target duplicates in the cache by comparing the types of the receiver and the types of the target. Figure 3 shows a call-site that has two different receiver types, `StayConstraint` and `EditConstraint`, which means that its lookup cache would have two entries: one for `StayConstraint#is_satisfied` and one for `EditConstraint#is_satisfied`. However, these two receivers resolve to the same target type `UnaryConstraint`: this target duplication in the cache can be avoided, monomorphizing the cache. We consider such a case to be relevant only when it has an impact on the lookup cache status: usually, it helps to reduce the degree of polymorphism (see Section 4.2).

In Figure 4, we identify the call-site `./deltablue.rb:47` as split: it is tied to two different virtual call-site IDs, which means it has been split once. This allows us to determine the status and evolution of lookup caches, distinguishing different virtual call-sites (see Section 4.5).

Bootstrap and Application phases. Initialization phases often differ from the rest of the execution, for instance, because objects and classes need to be loaded and initialized. In our analysis, we distinguish bootstrap and application phase. During the bootstrap phase, the TruffleRuby core libraries are loaded and initialized. The application phase starts when the runtime starts to load and execute the user code. We analyze the impact of the bootstrap phase on call-site behavior in Section 4.6. Where not otherwise stated, we include data from both phases.

A case of receiver polymorphism

```
493   ./deltablue.rb:47 execute EditConstraint EditConstraint 1397 1894
499   ./deltablue.rb:47 execute EqConstraint EqConstraint 1397 1894
518   ./deltablue.rb:47 execute EditConstraint EditConstraint 1397 1894
```

Figure 2. The lexical call-site at line 47 of the file `./deltablue.rb` is polymorphic with two entries for `EditConstraint` and `EqConstraint`. The call logged on line 518 did not modify the cache state.

A case of target duplication in the cache

```
3023  ./deltablue.rb:75 is_satisfied StayConstraint UnaryConstraint 1093 2368
3054  ./deltablue.rb:75 is_satisfied EditConstraint UnaryConstraint 1093 2368
```

Figure 3. This call-site would be considered polymorphic if we only considered the type of the receivers: `StayConstraint` and `EditConstraint`. The call-site can be monomorphized with the knowledge that both receivers resolve to the same type of target `UnaryConstraint`.

A case of splitting

```
3233   ./deltablue.rb:47 each Plan Vector 1031 3523
3895   ./deltablue.rb:47 each Plan Vector 1138 3523
```

Figure 4. This call-site has been split, as evidenced by the two different call-site IDs `1031` and `1138`. It means that the lexical call-site `./deltablue.rb:47` is linked to two virtual call-sites.

[4] The instrumented TruffleRuby and the analysis scripts in R are available at https://github.com/sophie-kaleba/ruby-cs-analyser.

4 Results

Our analysis first examines the general metrics of the benchmark set. It then focuses on those with megamorphic call-sites, as well as on those displaying a high proportion of polymorphic call-sites, to detail the impact of optimizations, call-site behavior over time, and the differences in use of methods and closures.

4.1 General Metrics of the Benchmarks

As mentioned before, we aim to include larger real-world-inspired workloads in our analysis.

To measure the size of the code in question, instead of taking a static line count, we count the number *statements* (Stmts) that are loaded during execution as well as the number of statements that were executed. This means we also count the code of Ruby's standard library, but ignore code that is not loaded or executed.

The first columns Stmts and Stmts Cov. in Table 1 show the number of statements per benchmark and the fraction executed. The columns Fns and Fns Cov. give the number of functions per benchmark and the fraction of these functions that were executed. The column kCalls contains the number of calls performed at run time, followed by the proportion of these calls that were polymorphic or megamorphic. The last two columns give the corresponding static count of call-sites. As can be seen in this table, with BlogRails and ERubiRails, we included benchmarks with nearly 120,000 statements, both of which execute about 45% of the statements at least once. The next largest benchmark is MailBench with about 32,000 statements of which 40.2% are executed. The smallest benchmark is Sieve. While it has merely 26 lines with statements, Ruby loads files with 15,699 statements of which 29% are executed at least once. The remaining benchmarks are in the range of about 15,000 to 27,000 statements, with around 30-40% of the statements being executed.

For all benchmarks the coverage of functions and methods is in the range of 19-38%. The Rails benchmarks are the largest with around 37,500 functions of which 35-38% are executed.

From the perspective of call-site behavior, we can already see in the Poly+Mega call-sites column that the larger benchmarks usually have a larger number of call sites that are polymorphic. On the other hand, many of the classic benchmarks have very few polymorphic call-sites (usually less than 1%).

Research Question 1. *Can these benchmarks be divided in distinct sets based on their call-site behavior?*

Observation 1. *Based on the polymorphism of call-sites that can be seen in Table 1, we can divide the benchmarks into three sets.* ***Megamorphic benchmarks*** *have at least one call-site with more than eight receivers (the size of a TruffleRuby cache);* ***polymorphic*** *benchmarks have more than 1.5% of their call-sites polymorphic; and the remaining benchmarks are* ***minimally-polymorphic****.*

The tables show the benchmark set in the order of to these three categories: the first category contains the megamorphic benchmarks e.g. BlogRails. See also Table 2 indicating the megamorphic calls directly. Next is the polymorphic category with 5 benchmarks including DeltaBlue, and lastly at the bottom of the table are the minimally-polymorphic benchmarks. By separating the benchmarks, we can focus on the ones with stronger polymorphism. Thus, we will exclude from now on the *minimally polymorphic* benchmarks from the tables, i.e. Are We Fast Yet's microbenchmarks, a number of image-processing benchmarks, and smaller benchmarks from the TruffleRuby benchmarks. The full tables can be found in Appendix A and B.

Research Question 2. *What proportion of call-sites are monomorphic?*

Observation 2. *In BlogRails and ERubiRails, 97.7% of the call-sites are monomorphic. They have the largest code-bases of our set and most of the polymorphism can be explained by code re-use, e.g. use of libraries.*

In Sinatra and HexaPdf, which are smaller than BlogRails and ERubiRails, around 95.7% of call-sites are monomorphic. For the other benchmarks, about 98.2% *are monomorphic.*

The results for the other benchmarks, with about 98.2% being monomorphic, are in line with previous reports in the literature, which found about 98% of the call-sites in large benchmarks to be monomorphic[17].

4.2 Impact of the Existing Call-Site Optimizations on Call-Site Behavior

As described in Section 2.3 and 2.5, TruffleRuby uses splitting and eliminates call target duplication to optimize call-sites. Since previous studies have not considered how effective these optimizations are in combination, we ask the following research questions:

Research Question 3. *To what degree does eliminating call target duplicates reduce polymorphism?*

Research Question 4. *To what degree does splitting reduce polymorphism after eliminating call target duplicates?*

To answer these questions, we focus on the benchmarks in our megamorphic and polymorphic category and check first, whether the caches contain target duplicates, and then whether eliminating the duplicates leads to a smaller number of polymorphic calls. Table 2 shows the results: for each benchmark, it lists the number of polymorphic and megamorphic calls before eliminating any target duplicates (columns two and three). It also lists the percentage by which the number of polymorphic and megamorphic calls decreased because of the fewer entries in the caches after removing

Table 1. Around 38% of all statements including libraries are executed, which equates to a 27% coverage across all methods. 46 benchmarks are megamorphic, from a total of 74 benchmarks: the majority of these are industrial benchmarks. The *-suffixed benchmarks have been aggregated due to their similar behavior, and their values have been averaged.

Benchmark	Stmts	Stmts Cov.	Fns	Fns Cov.	kCalls	Poly+ Mega. calls	Exec. call-sites	Poly+ Mega. call-sites
BlogRails	118,717	48%	37,595	38%	13,863	7.4%	52,361	2.3%
ChunkyCanvas*	19,279	32%	5,082	20%	11,323	0.0%	1,816	1.0%
ChunkyColor*	19,266	32%	5,077	20%	19	2.0%	1,790	1.0%
ChunkyDec	19,289	32%	5,083	20%	21	2.0%	1,809	1.2%
ERubiRails	117,922	45%	37,328	35%	12,309	5.4%	47,794	2.3%
HexaPdfSmall	26,624	44%	6,990	35%	31,246	7.4%	6,872	4.1%
LiquidCartParse	23,531	37%	6,259	27%	87	1.3%	3,065	1.9%
LiquidCartRender	23,562	39%	6,269	30%	236	5.5%	3,581	2.4%
LiquidMiddleware	22,374	37%	5,939	27%	70	1.4%	2,918	1.4%
LiquidParseAll	23,276	37%	6,186	27%	295	1.9%	3,127	2.2%
LiquidRenderBibs	23,277	39%	6,185	29%	385	23.4%	3,466	2.8%
MailBench	31,857	40%	8,392	32%	2,756	3.4%	5,414	3.6%
PsdColor	27,498	40%	7,724	28%	352	4.1%	6,668	1.9%
PsdCompose*	27,498	40%	7,724	28%	352	4.0%	6,678	2.0%
PsdImage*	27,531	40%	7,736	28%	5,509	0.0%	6,677	2.0%
PsdUtil*	27,496	40%	7,724	28%	351	4.0%	6,655	2.0%
Sinatra	31,187	40%	8,492	29%	172	6.9%	5,639	4.4%
ADConvert	21,588	37%	4,771	27%	371	7.9%	3,979	3.1%
ADLoadFile	21,586	35%	4,771	26%	171	13.2%	3,335	2.9%
DeltaBlue	16,292	31%	4,052	21%	13	6.4%	1,738	2.4%
PsychLoad	19,282	36%	4,982	25%	6,232	11.6%	2,412	1.9%
RedBlack	15,909	30%	3,915	20%	42,897	20.3%	1,774	2.9%
Acid	15,703	29%	3,877	19%	9	1.7%	1,445	0.7%
BinaryTrees	15,708	30%	3,876	20%	6,355	0.0%	1,474	0.7%
Bounce	15,979	29%	3,953	19%	16	0.9%	1,457	0.7%
CD	16,386	30%	4,025	20%	75,184	6.2%	1,772	0.7%
Fannkuch	15,729	30%	3,873	19%	10,864	0.0%	1,473	0.7%
Havlak	16,237	31%	4,027	21%	44,901	3.0%	1,710	0.7%
ImgDemoConv	15,776	29%	3,905	20%	3,417	0.0%	1,512	0.7%
ImgDemoSobel	15,818	30%	3,920	20%	3,806	0.0%	1,518	0.7%
Json	16,223	30%	4,024	20%	210	0.1%	1,584	0.6%
List	15,716	29%	3,878	19%	53	0.3%	1,457	0.7%
Mandelbrot	15,730	29%	3,872	19%	9	1.7%	1,437	0.7%
MatrixMultiply	15,712	29%	3,879	20%	100	0.1%	1,473	0.7%
NBody	15,763	29%	3,892	19%	9	1.6%	1,518	0.7%
NeuralNet	15,792	30%	3,911	20%	33,010	0.0%	1,602	0.7%
OptCarrot	18,518	35%	4,450	24%	9,242	0.0%	2,544	1.0%
Permute	15,707	29%	3,875	19%	40	0.4%	1,445	0.7%
Pidigits	15,714	29%	3,873	19%	97	0.2%	1,456	0.7%
Queens	15,716	29%	3,878	19%	23	0.6%	1,449	0.7%
Richards	15,935	30%	3,934	20%	1,553	0.0%	1,584	0.6%
Sieve	15,699	29%	3,873	19%	9	1.7%	1,440	0.7%
SpectralNorm	15,715	29%	3,882	20%	6,441	0.0%	1,479	0.7%
Storage	15,954	29%	3,950	19%	24	0.6%	1,449	0.7%
Towers	15,726	29%	3,882	19%	82	0.2%	1,456	0.7%

the duplicates (columns four and five). Appendix B provides additional data from a static perspective, showing the impact of these optimizations on the number of polymorphic and megamorphic call-sites.

Our results show that up to 93.6% of polymorphic calls are subject to target polymorphism. Eliminating target duplication reduces the number of polymorphic calls significantly. The benchmarks that benefit most are OptCarrot, a NES emulator, and LiquidParseAll, which parses Liquid HTML templates. OptCarrot had 93.6% of its polymorphic calls monomorphized. LiquidParseAll had 87.4% of its calls monomorphized. All of its megamorphic calls have been eliminated in the process as well.

Almost all of the megamorphic benchmarks see their megamorphic calls turning either polymorphic or even monomorphic. Four megamorphic benchmarks still experience megamorphic calls after the elimination of target duplicates: the two Ruby-on-Rails-based benchmarks BlogRails and ERubiRails, where the megamorphism is in the HTTP request routing and in the ActiveSupport library callbacks; HexaPdfSmall when validating PDF objects during the PDF generation process; and Sinatra, when compiling new HTTP routing paths.

Table 2. Eliminating target duplicates in the cache is very effective at reducing polymorphism: It is generally reduced by around 45%, except for RedBlack and CD that has less than 8% of duplicates

	Number of calls		After eliminating target duplicates	
Benchmark	Poly.	Mega.	Poly.	Mega.
BlogRails	956,515	63,319	-48.8%	-99.1%
ChunkyCanvas*	322	98	-80.0%	-100.0%
ChunkyColor*	320	98	-79.0%	-100.0%
ChunkyDec	322	98	-79.5%	-100.0%
ERubiRails	626,535	40,699	-37.4%	-98.6%
HexaPdfSmall	1,842,665	479,399	-21.7%	-99.6%
LiquidCartParse	821	280	-73.3%	-100.0%
LiquidCartRender	12,598	280	-84.1%	-100.0%
LiquidMiddleware	747	251	-68.8%	-100.0%
LiquidParseAll	5,369	280	-87.4%	-100.0%
LiquidRenderBibs	89,866	280	-73.7%	-100.0%
MailBench	81,886	12,697	-77.6%	-100.0%
PsdColor	14,053	233	-53.1%	-100.0%
PsdCompose*	14,053	233	-53.0%	-100.0%
PsdImage*	14,062	233	-53.0%	-100.0%
PsdUtil*	14,048	233	-53.0%	-100.0%
Sinatra	7,909	3,911	-82.8%	-94.4%
ADConvert	29,337	0	-58.3%	0.0%
ADLoadFile	22,654	0	-53.5%	0.0%
DeltaBlue	846	0	-33.7%	0.0%
PsychLoad	723,984	0	-85.7%	0.0%
RedBlack	8,718,802	0	-7.7%	0.0%

Observation 3. *We conclude that eliminating target duplicates reduces polymorphism successfully, eliminating megamorphic calls almost completely.*

Splitting eliminates almost all remaining polymorphism. Table 3 is structured similarly to Table 2. The table shows how splitting impacts polymorphism once target duplicates have been eliminated. The *Number of splits*-column indicates the number of method copies created. Notably, the table shows that all remaining polymorphic calls are monomorphized by splitting.

The benchmarks with a large number of polymorphic call-sites (see Table 11) usually have a high amount of splitting, as we would expect considering Truffle's splitting heuristic (see Section 2.3). For example, BlogRails and ERubiRails rank respectively first and second in terms of the number of times splitting occurred (2163 and 1851 times).

The minimally-polymorphic benchmarks excluded from Table 3 mostly behave homogeneously: with the exception of CD and Havlak, our two outliers with a large number of (polymorphic) calls, they all had around thirty polymorphic calls remaining, stemming from less than eight call-sites that were monomorphized by splitting (see Appendix B). Considering this small number of polymorphic call-sites remaining, the amount of splitting they experience is however high, with at least 27 splits occurring. In the following Section 4.3, we discuss why these benchmarks may experience splitting

more than twice, even though they had only a few remaining polymorphic call-sites.

Table 3. The polymorphic and megamorphic calls remaining after having eliminated target duplicates are almost completely monomorphized by splitting.

	Number of calls		After splitting		Number
Benchmark	Poly.	Mega.	Poly.	Mega.	of splits
BlogRails	490,072	557	-100%	-100%	2163
ChunkyCanvas*	66	0	-100%	0%	43
ChunkyColor*	66	0	-100%	0%	42
ChunkyDec	66	0	-100%	0%	42
ERubiRails	391,997	553	-100%	-100%	1851
HexaPdfSmall	1,443,211	2,066	-100%	-100%	498
LiquidCartParse	219	0	-100%	0%	107
LiquidCartRender	2,000	0	-100%	0%	207
LiquidMiddleware	233	0	-100%	0%	114
LiquidParseAll	679	0	-100%	0%	136
LiquidRenderBibs	23,633	0	-100%	0%	191
MailBench	18,322	0	-100%	0%	343
PsdColor	6,586	0	-100%	0%	300
PsdCompose*	6,586	0	-100%	0%	300
PsdImage*	6,588	0	-100%	0%	300
PsdUtil*	6,584	0	-100%	0%	300
Sinatra	1,362	220	-100%	-100%	297
ADConvert	12,226	0	-100%	0%	236
ADLoadFile	10,525	0	-100%	0%	175
DeltaBlue	561	0	-100%	0%	78
PsychLoad	103,506	0	-100%	0%	78
RedBlack	8,043,472	0	-100%	0%	50

Tables 2 and 3 provide a dynamic perspective on the impact of eliminating target duplicates and splitting on call-site behavior. We show these two optimizations are very successful at monomorphizing polymorphic call-sites, as well as eliminating megamorphism.

Table 4 shows how the two optimizations influence the maximum number of targets per cache. The larger benchmarks have a higher maximum number of receivers for at least one call site (i.e. the Ruby-on-Rails benchmarks with at least one cache holding 206 targets). Furthermore, eliminating duplicates significantly decreases the degree of polymorphism in the megamorphic benchmarks subset, only few caches remain polymorphic. Similarly to what we observed in Table 2, only the most megamorphic benchmarks such as the two Ruby-on-Rails benchmarks and Sinatra stay megamorphic before splitting is considered. HexaPdfSmall stays megamorphic before splitting, even so the maximum cache size is relatively small with only 11 targets. The minimally-polymorphic benchmarks, not pictured here, all behave homogeneously, with a maximum of four targets before all optimizations, reduced to two after eliminating duplicates, and completely monomorphized after splitting. A closer look at the distribution of receivers shows that the distribution remains unchanged at any optimization stage, with at least 75% of calls being monomorphic only.

Observation 4. *Splitting in combination with addressing target polymorphism is effective at monomorphizing polymorphic call-sites. Only two benchmarks from our set still display polymorphism, with caches containing at most two targets. All other benchmarks have been completely monomorphized.*

Table 4. Eliminating target duplicates, in addition to splitting, reduces the maximum cache size. Both optimizations together turn almost all caches monomorphic, even when they held many targets initially.

	Biggest cache size (number of targets)		
Benchmark	before all optimisations	after eliminating duplicates	after splitting
BlogRails	206	24	2
ChunkyCanvas*	15	2	1
ChunkyColor*	15	2	1
ChunkyDec	15	2	1
ERubiRails	206	24	2
HexaPdfSmall	20	11	1
LiquidCartParse	20	2	1
LiquidCartRender	20	5	1
LiquidMiddleware	18	2	1
LiquidParseAll	20	4	1
LiquidRenderBibs	20	7	1
MailBench	71	3	1
PsdColor	31	3	1
PsdCompose*	31	3	1
PsdImage*	31	3	1
PsdUtil*	31	3	1
Sinatra	84	16	1
ADConvert	8	2	1
ADLoadFile	7	2	1
DeltaBlue	4	3	1
PsychLoad	5	3	1
RedBlack	4	2	1

4.3 Splitting Transitions

There are many possible changes of lookup cache state after splitting, but ideally it leads to a lower degree of polymorphism. Thus, our question is:

Research Question 5. *What are the most frequent lookup cache state transitions after splitting?*

Hölzle et al. [9] stated that the aim of splitting is to monomorphize polymorphic call-sites. Indeed, Truffle's heuristic (see Section 2.3) will mark a method as candidate for splitting as soon as a lookup cache gains a second entry.

Two cases are therefore possible:

- The clone's cache contains the same target as the original, which means that the call-site would have remained monomorphic if it had not been split, suggesting the split that occurred was unnecessary;
- The clone's cache contains a different target, which means that if splitting had not been triggered, the cache would have turned polymorphic.

Table 5 shows the frequency of these splitting transitions for our benchmark set: it displays the number of splitting actions, and how splitting influenced the lookup cache state of the split call-sites. Quite unexpectedly column four indicates that 89% of splitting happens on monomorphic call-sites that remain monomorphic with the same target after splitting, which suggests over-splitting may have occurred. We inspected several of these cases manually and identified that they result from recursive splitting (see Section 2.3).

Observation 5. *The most common splitting outcome results in the clone's cache containing the same entry as the original cache. In significantly fewer cases splitting prevented a cache*

Table 5. State transition of lookup caches for method call-sites. The results indicate that 88.7% of splitting results in call-sites having the same single entry as before the split, which may indicate a too aggressive splitting strategy.

		Cache entries after splitting (% of total number of splits)	
Benchmark	**Number of splits**	**Different**	**Same**
BlogRails	2163	14%	86%
ChunkyCanvas*	43	7%	93%
ChunkyColor*	42	7%	93%
ChunkyDec	42	7%	93%
ERubiRails	1851	15%	85%
HexaPdfSmall	498	9%	91%
LiquidCartParse	107	7%	93%
LiquidCartRender	207	8%	92%
LiquidMiddleware	114	10%	90%
LiquidParseAll	136	6%	94%
LiquidRenderBibs	191	12%	88%
MailBench	343	7%	93%
PsdColor	300	11%	89%
PsdCompose*	300	11%	89%
PsdImage*	300	11%	89%
PsdUtil*	300	11%	89%
Sinatra	297	10%	90%
ADConvert	236	11%	89%
ADLoadFile	175	11%	89%
DeltaBlue	78	28%	72%
PsychLoad	78	8%	92%
RedBlack	50	40%	60%
Acid	27	7%	93%
BinaryTrees	30	7%	93%
Bounce	27	7%	93%
CD	41	12%	88%
Fannkuch	27	7%	93%
Havlak	45	7%	93%
ImgDemoConv	30	7%	93%
ImgDemoSobel	27	7%	93%
Json	27	7%	93%
List	27	7%	93%
Mandelbrot	27	7%	93%
MatrixMultiply	31	6%	94%
NBody	28	7%	93%
NeuralNet	46	7%	93%
OptCarrot	66	12%	88%
Permute	28	7%	93%
Pidigits	27	7%	93%
Queens	28	7%	93%
Richards	29	7%	93%
Sieve	27	7%	93%
SpectralNorm	30	7%	93%
Storage	27	7%	93%
Towers	27	7%	93%

from turning polymorphic: this is the case when the clone's cache and the original cache contain different entries. As previously mentioned, the splitting strategy in TruffleRuby might be overly aggressive.

4.4 Methods Versus Closures

Ruby provides both methods and closures to the developer (see Section 2.4). While we have focused so far only on methods, this section analyses the call-site behavior of closures, asking the question:

Research Question 6. *Do the call-site behaviors of methods and closures differ?*

Table 6 summarizes the polymorphic behavior of closures in our benchmarks. Columns 2 to 4 repeat the general metrics about methods already shown in Table 1. Columns 5 to 8 mirror these metrics for closures. The kCalls column contains the total number of times a method or a closure has been called. The Poly. calls column to the right of it represents the fraction of these calls that are polymorphic and megamorphic. The same structure applies to the Exec. call-sites, representing the total number of method or closure call-sites, and the Poly. call-sites column on its right.

Table 6. Around 96.5% of closure call-sites are monomorphic, with ERubiRails being the least polymorphic benchmark with 2.1%, and Havlak the most polymorphic one with 8.5% of polymorphic closure call-sites.

	METHODS				CLOSURES			
Benchmark	**kCalls**	**Poly+ Mega. calls**	**Exec. call-sites**	**Poly+ Mega. call-sites**	**kCalls**	**Poly+ Mega. calls**	**Exec. call-sites**	**Poly.+ Mega call-sites**
BlogRails	13,863	7.4%	52,361	2.3%	1,410	10.1%	10,026	2.3%
ChunkyCanvas*	11,323	0.0%	1,816	1.0%	3,133	26.0%	178	7.0%
ChunkyColor*	19	2.0%	1,790	1.0%	2	12.0%	175	6.0%
ChunkyDec	21	2.0%	1,809	1.2%	2	9.9%	177	6.2%
ERubiRails	12,309	5.4%	47,794	2.3%	1,100	5.3%	9,314	2.1%
HexaPdfSmall	31,246	7.4%	6,872	4.1%	3,237	2.3%	945	5.8%
LiquidCartParse	87	1.3%	3,065	1.9%	7	5.7%	272	7.0%
LiquidCartRender	236	5.5%	3,581	2.4%	14	5.0%	360	7.2%
LiquidMiddleware	70	1.4%	2,918	1.4%	6	6.2%	263	5.7%
LiquidParseAll	295	1.9%	3,127	2.2%	12	11.1%	284	7.0%
LiquidRenderBibs	385	23.4%	3,466	2.8%	34	7.7%	408	5.9%
MailBench	2,756	3.4%	5,414	3.6%	124	3.9%	628	4.1%
PsdColor	352	4.1%	6,668	1.9%	90	2.5%	923	3.5%
PsdCompose*	352	4.0%	6,678	2.0%	90	2.0%	925	4.0%
PsdImage*	5,509	0.0%	6,677	2.0%	1,934	0.0%	925	4.0%
PsdUtil*	351	4.0%	6,655	2.0%	90	2.0%	922	4.0%
Sinatra	172	6.9%	5,639	4.4%	23	7.9%	855	4.0%
ADConvert	371	7.9%	3,979	3.1%	31	4.4%	335	6.3%
ADLoadFile	171	13.2%	3,335	2.9%	14	6.6%	235	7.7%
DeltaBlue	13	6.4%	1,738	2.4%	2	13.5%	207	5.8%
PsychLoad	6,232	11.6%	2,412	1.9%	381	0.1%	217	6.9%
RedBlack	42,897	20.3%	1,774	2.9%	801	25.0%	145	6.2%
Acid	9	1.7%	1,445	0.7%	1	22.6%	130	6.9%
BinaryTrees	6,355	0.0%	1,474	0.7%	23	1.0%	136	6.6%
Bounce	16	0.9%	1,457	0.7%	6	4.4%	131	6.9%
CD	75,184	6.2%	1,772	0.7%	707	0.3%	146	8.2%
Fannkuch	10,864	0.0%	1,473	0.7%	1	22.6%	134	6.7%
Havlak	44,901	3.0%	1,710	0.7%	7,435	4.3%	176	8.5%
ImgDemoConv	3,417	0.0%	1,512	0.7%	130	0.2%	138	6.5%
ImgDemoSobel	3,806	0.0%	1,518	0.7%	217	0.1%	140	6.4%
Json	210	0.1%	1,584	0.6%	1	19.3%	130	6.9%
List	53	0.3%	1,457	0.7%	1	22.7%	129	7.0%
Mandelbrot	9	1.7%	1,437	0.7%	1	22.5%	129	6.2%
MatrixMultiply	100	0.1%	1,473	0.7%	3,444	0.0%	141	7.1%
NBody	9	1.6%	1,518	0.7%	1	24.1%	134	8.2%
NeuralNet	33,010	0.0%	1,602	0.7%	5,541	0.2%	199	6.0%
OptCarrot	9,242	0.0%	2,544	1.0%	4,561	47.4%	311	5.5%
Permute	40	0.4%	1,445	0.7%	6	3.7%	130	7.7%
Pidigits	97	0.2%	1,456	0.7%	9	2.4%	130	6.9%
Queens	23	0.6%	1,449	0.7%	10	11.4%	130	6.9%
Richards	1,553	0.0%	1,584	0.6%	76	86.8%	131	7.6%
Sieve	9	1.7%	1,440	0.7%	6	3.7%	130	6.9%
SpectralNorm	6,441	0.0%	1,479	0.7%	6,417	0.0%	141	7.8%
Storage	24	0.6%	1,449	0.7%	6	3.4%	130	6.9%
Towers	82	0.2%	1,456	0.7%	1	23.9%	129	7.8%

Method calls are much more frequent than closure calls, by a factor of around 7 in all of our benchmarks. However closure call-sites are also more likely to be polymorphic than method call-sites, even though there are many fewer. The only exceptions are the two Ruby-on-Rails benchmarks: ERubiRails, which has the fewest polymorphic closure call-sites of the set, with 2.1% of closure call-sites being polymorphic, and BlogRails with 2.3% polymorphic closure call-sites. Havlak ranks first with 8.5% of closure call-sites polymorphic. None of the benchmarks are minimally polymorphic if considering polymorphism for closure call-sites.

With the most polymorphic benchmark having only 4.4% of *method* call-sites being polymorphic, a slightly high proportion of closure than method call-sites is polymorphic.

By default, TruffleRuby used to force the splitting of closures when a new closure was to be added to the lookup cache. This hides the impact of splitting from our analysis. Therefore, we disabled this option so that the splitting decisions are only guided by Truffle's heuristics (see Section 2.3), which are also used for methods. In this context, split-induced monomorphization is still frequent and effective, leading to a complete elimination of the polymorphism in closure call-sites, as shown in the four last columns of Table 7.

Table 7. Splitting succeeds at completely monomorphizing the existing polymorphic closure calls, in a similar fashion to method calls.

	METHODS		CLOSURES			
	After eliminating duplicates and splitting		Before splitting (number of calls)		After Splitting (% change)	
Benchmark	Poly.	Mega.	Poly.	Mega.	Poly.	Mega.
BlogRails	-100%	-100%	142,222	224	-100%	-100%
ChunkyCanvas*	-100%	-100%	2,505,980	0	-100%	0%
ChunkyColor*	-100%	-100%	230	0	-100%	0%
ChunkyDec	-100%	-100%	230	0	-100%	0%
ERubiRails	-100%	-100%	58,190	224	-100%	-100%
HexaPdfSmall	-100%	-100%	75,669	0	-100%	0%
LiquidCartParse	-100%	-100%	411	0	-100%	0%
LiquidCartRender	-100%	-100%	710	0	-100%	0%
LiquidMiddleware	-100%	-100%	344	0	-100%	0%
LiquidParseAll	-100%	-100%	1,360	0	-100%	0%
LiquidRenderBibs	-100%	-100%	2,634	0	-100%	0%
MailBench	-100%	-100%	4,848	0	-100%	0%
PsdColor	-100%	-100%	2,209	0	-100%	0%
PsdCompose*	-100%	-100%	2,206	0	-100%	0%
PsdImage*	-100%	-100%	2,463	0	-100%	0%
PsdUtil*	-100%	-100%	2,205	0	-100%	0%
Sinatra	-100%	-100%	1,837	0	-100%	0%
ADConvert	-100%	0%	1,366	0	-100%	0%
ADLoadFile	-100%	0%	923	0	-100%	0%
DeltaBlue	-100%	0%	268	0	-100%	0%
PsychLoad	-100%	0%	257	0	-100%	0%
RedBlack	-100%	0%	200,221	0	-100%	0%
Acid	-100%	0%	223	0	-100%	0%
BinaryTrees	-100%	0%	223	0	-100%	0%
Bounce	-100%	0%	273	0	-100%	0%
CD	-100%	0%	2,170	0	-100%	0%
Fannkuch	-100%	0%	223	0	-100%	0%
Havlak	-100%	0%	319,468	0	-100%	0%
ImgDemoConv	-100%	0%	223	0	-100%	0%
ImgDemoSobel	-100%	0%	223	0	-100%	0%
Json	-100%	0%	223	0	-100%	0%
List	-100%	0%	223	0	-100%	0%
Mandelbrot	-100%	0%	221	0	-100%	0%
MatrixMultiply	-100%	0%	971	0	-100%	0%
NBody	-100%	0%	247	0	-100%	0%
NeuralNet	-100%	0%	10,240	0	-100%	0%
OptCarrot	-100%	0%	2,163,026	0	-100%	0%
Permute	-100%	0%	225	0	-100%	0%
Pidigits	-100%	0%	223	0	-100%	0%
Queens	-100%	0%	1,109	0	-100%	0%
Richards	-100%	0%	66,017	0	-100%	0%
Sieve	-100%	0%	223	0	-100%	0%
SpectralNorm	-100%	0%	1,433	0	-100%	0%
Storage	-100%	0%	223	0	-100%	0%
Towers	-100%	0%	238	0	-100%	0%

Observation 6. *The polymorphic behavior of closures and methods is similar, but methods calls are more frequent, and closure calls tend to be slightly more polymorphic.*

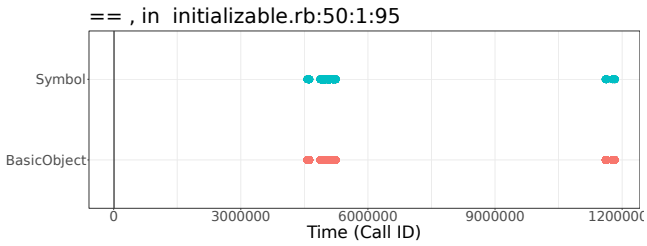


Figure 5. Mirroring pattern in ERubiRails. The plot shows calls over time (Call ID). Calls with targets Symbol and BasicObject exhibit the same behavior, i.e., mirror each other.

4.5 Lookup Cache State Evolution Throughout Execution

As we saw in the previous sections, the polymorphism of call-sites is drastically reduced by eliminating target duplicates and by splitting. However, our results indicate that there may be over-splitting.

One source of polymorphism may be that an application has a distinct initialization phase, where initialization and setup code is executed. This may leave lookup cache entries behind for types that are not used later on anymore. To investigate more broadly, we ask the following question:

Research Question 7. *What kind of behavioral patterns do call-sites exhibit?*

To answer this question, we analyze the behavior of our 51 most polymorphic benchmarks (see Tables 2 and 3) just after having eliminated target duplicates in the cache. Splitting is disabled, since it would fully monomorphize the call-sites. We narrowed our investigation down to the call sites that have target polymorphism and ignored call-sites that have fewer than 10 calls. We plot the calls to the different targets over time, which we derive from the line number in the log file (see Section 3.2). The vertical black line delimits the switch from bootstrap to application phase (see Section 3.2). We can categorize them into three distinct call-site behavior patterns. that repeat across our benchmarks.

Mirroring pattern. Different targets at a call-site may display similar behavior throughout execution. Figure 5 shows an example for the ERubiRails benchmark, which renders an ERB template. A call to an `==` method sees two targets. The implementations in the `Symbol` and `BasicObject` classes are called in a very similar pattern.

Phase behavior. Phase behavior is characterized by a visible change in the pattern over time. We distinguish here two variants, behavior at the level of a single target and behavior at the level of the call-site. Possible phase changes include a distinct change in call frequency as well as changes in which targets are called at a call-site.

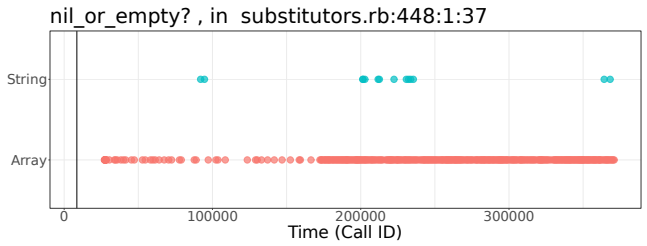


Figure 6. Phase behavior for the Array target in ADConvert. The frequency of calls to this target increases noticeably about halfway through the benchmark.

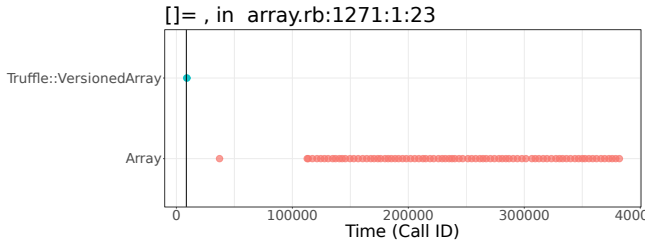


Figure 7. Initialization pattern. This is an extreme example with a tiny number of calls to VersionedArray initially, after which only Array is called.

Figure 6 shows an example for a call site with a distinct change halfway through the execution: at this point the frequency of calls on `Array` objects significantly increases.

Other call-sites may go from being monomorphic to polymorphic, or the other way around, i.e., only a single target remains relevant after the phase change.

Initialization pattern. The Initialization pattern is a specific example of the Phase behavior pattern. In this pattern, calls are made to certain call-targets at the beginning of execution until a certain distinct point is reached, after which the behavior changes, for instance to use a different set of targets. Figure 7 shows an extreme example where a tiny number of calls is made to the `VersionedArray` class initially, after which all calls go to `Array`.

No apparent pattern. Some benchmarks have polymorphic call-sites that do not display any apparent pattern or phases in their behavior. An example is shown in Figure 8. Here both targets have slightly different frequency and timings, but do not divide the execution into clear phases or exhibit clear patterns.

Observation 7. *As others have observed [14, 15], some call-sites exhibit an initialization pattern in our benchmarks. In addition, we see patterns of mirrored behavior, phase behavior, and call-sites without any apparent pattern.*

Some call-sites exhibit several of these patterns at once. This is the case for instance for the `execute` call-site in the DeltaBlue benchmark shown in Figure 9. It displays partial

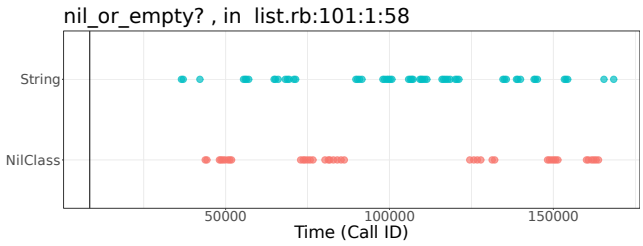


Figure 8. No Apparent pattern. This call-site in the ADLoadFile benchmark displays no apparent pattern.

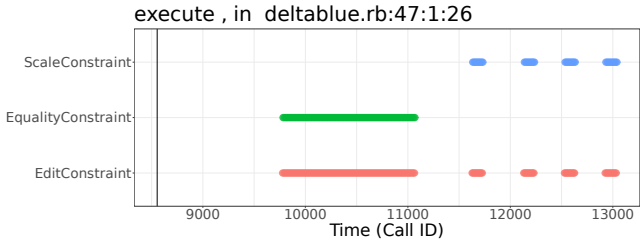


Figure 9. Multiple patterns. This call-site in the DeltaBlue benchmark displays several call-site behavior patterns at once.

mirroring across the three targets and shows phase behavior, switching halfway through.

4.6 Impact of Bootstrap

We can see in the plots of Section 4.5 that the bootstrap process does not cause extra polymorphism for frequently used call targets. However, these plots represent only hot call-sites, which brings us to our next question:

Research Question 8. *Does the bootstrap process of TruffleRuby influence the degree of polymorphism in our benchmark set?*

To answer this question, we first identify the call-sites called during the bootstrap phase. If they are later called during the application phase, we check whether the content of their lookup caches differ. If this is the case, it means that the bootstrap phase causes polymorphism.

In our benchmark set, the bootstrap phase is only having minimal impact on the degree of polymorphism. Only four call-sites see an increase of their cache size. However, it concerns 51 of our benchmarks, especially the megamorphic and polymorphic ones. Three of these call-sites see their degree of polymorphism increased by one, which means that one target was added in the cache during the bootstrap phase, but not reused later during the application phase. These call-sites are related to module loading and implicit type conversion. One call-site sees its degree of polymorphism increased by two; it is used to resolve classes in a polyglot context.

In Ruby, class loading and initialization can happen on demand. This blurs the boundaries between initialization and computation phases and makes it harder to consistently identify the switch between the two. Bootstrapping on the other hand is finished once the use code starts executing. However, the small amount of call-sites impacted by the bootstrap phase suggest that one may need to look at a more comprehensive initialization phase, too.

Observation 8. *With only four call sites seeing their degree of polymorphism increased, the loading of TruffleRuby's core libraries increases the degree of polymorphism only minimally in our benchmark set.*

4.7 Limitations

In this section, we discuss some limitations of our study.

Core library calls differentiation. Some core library calls and builtins are not associated with a lexical location. We discard these cases since we use the lexical location to uniquely identify call-sites.

Splitting specializations. TruffleRuby splits method call-sites once they turn polymorphic. However, methods are also split when there are other types of polymorphic operations, e.g., addition operators acting on integers and doubles, or string operations seeing different string encodings. These use-cases were outside the scope of our study as we consider only call-site behavior, and are therefore ignored by our instrumentation. However, we confirmed that only a negligible number of splitting operations are caused by these omitted operations.

5 Insight and Recommendations

We analyzed a diverse set of Ruby applications including large applications such as BlogRails and MailBench, as well as medium-sized and smaller benchmarks, used both for research and to optimize language implementations used in production. We found that the majority of our benchmarks display a low degree of polymorphism, with only 2% call-sites being polymorphic, which is roughly in line with the literature. However, we also see that our industrial benchmarks display more polymorphic behavior, which has been underrepresented in previous studies, and should be carefully considered when composing benchmark sets.

We find that when the elimination of call target duplicates is combined with splitting, all but a few call-sites were monomorphized (see Section 4.2). However, our findings also suggest that splitting may be too aggressive leading to unnecessary method duplication. This is problematic, because splitting has a run-time cost in terms of additional memory used for the split methods, additional cost in lookups (since split copies have uninitialized lookup caches) and profiling of the split copies, as well as additional just-in-time compilation costs, and thus, prolonged warmup. For large codebases, this can be a performance issue for just-in-time compilation.

While we might expect splitting most frequently to monomorphize polymorphic call-sites or prevent call-sites from turning polymorphic, it appears that most splitting is performed on monomorphic call-sites, and leaves the lookup cache in an identical state (see Section 4.3). Considering the impact this may have on both performance and memory consumption, this will require further analysis to identify whether it is avoidable.

One possible starting point to approach this issue is to investigate the evolution of call-site behavior and lookup cache usage during execution, with splitting is disabled. We identified three patterns across our polymorphic benchmarks (see Section 4.5). Knowledge of execution patterns has been used in previous research to guide optimizations [11], and we argue that the call-site patterns identified here may help guide the development of other call-site optimizations.

6 Related Work on Dynamic Program Behavior

Our work focuses on the call-site behavior of Ruby applications. To the best of our knowledge, little work has previously investigated this aspect. The most closely related work is by Åkerblom and Wrigstad [2], who measure polymorphism in Python programs. Similarly to our approach, they measure the degree of receiver polymorphism (see Section 4.1) and target polymorphism, which they refer to as n-typeable. They also aim to guide language developers to further optimize Python, but their focus is on type systems. Our work investigates how polymorphism evolves at call-sites, differences between method and closure calls, and the impact of optimizations on these.

Several other works have investigated dynamism, which inspires some of our methodology. Similar to the work by Åkerblom et al. [1] on dynamism in Python programs, we analyze the behavior of programs written in a dynamic language using a custom, instrumented version of the runtime. We also look at similar call-site behaviors. For instance, Richards et al. [16] investigate the validity of common assumptions about the dynamic behavior of JavaScript programs and notably show that call-site dynamism (i.e. polymorphism, as in different function bodies per call-site, see Section 2.3 in this paper) is more frequent than is commonly assumed. These works however do not consider the evolution of dynamism beyond the 'startup vs. steady state' dichotomy. The impact of existing optimizations is also not considered.

Holkner and Harland [8] also when and how Python programs use the more dynamic, reflection-like features, also distinguishing between startup and later run time. However, they do not consider the evolution of call-site behavior.

Sarimbekov et al. [17] analyzed call-site behavior for JVM languages. While they included Ruby, they use microbenchmarks and a single larger application per language, and do not analyze how call-site behavior evolves.

7 Conclusion and Future Work

We analyze the call-site behavior of a wide range of Ruby applications, focusing on the state of the lookup caches and how they are affected by optimizations at run time. Our results show that while most of our benchmarks are monomorphic, the larger ones, more representative of workloads of real applications, show more polymorphic behavior. We can confirm that lookup caches, splitting, and the elimination of target duplication are highly effective in monomorphizing call-sites. However, we also observe that splitting may be applied too often.

Future optimizations may be informed by the three patterns we found in the evolution of call-site behavior over time. Specifically, we confirm the initialization pattern, but also saw phased behavior. Furthermore, we saw mirror patterns, suggesting that call-sites may exhibit correlated behaviors; this may be exploitable in the future. Other call-sites displayed no immediately apparent behavior patterns.

We found that a slightly higher proportion of closure call-sites is polymorphic compared to method call-sites. However, at least in the case of TruffleRuby, we also noticed that the currently used forced splitting may again be overly aggressive, and a standard splitting strategy may suffice.

In future work, we want to investigate in more detail how calls relate to each other. Traditional inlining and the work of Flückiger et al. [6] show that calling context can be used to enable optimizations, and also to reduce call overhead. Since the performance of many large applications is dominated by calls, it seems relevant to investigate further how call relations can be exploited to reduce call overhead.

The patterns of call-site behavior we identified, such as the phase behavior and the initialization patterns, may be able to guide splitting to monomorphize call-sites without over-splitting, thus reducing the associated memory and execution overhead, currently a challenge for the warmup behavior of just-in-time compilers on large codebases. Such an analysis would ideally determine the perfect splitting for applications and compare this to current heuristics.

Another aspect not currently taken into account is polymorphism originating from data flow rather than call relations. The behavior of objects and closures accessed through arguments or fields at polymorphic call-sites have not been studied or considered in splitting heuristics.

The effective monomorphization also opens new doors for other run-time techniques that introduce polymorphism. For instance, tracking whether objects are only accessible by a single thread as part of the receiver type can introduce polymorphism [5]. Similarly, one may use receiver types to record data-race-related information for run-time race detection. In both cases, the effectiveness of eliminating target duplicates is essential to minimize overhead.

Acknowledgments

We thank Chris Seaton (Shopify) and Benoit Daloze (Oracle Labs) for feedback on an early draft.

This work was supported by the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

A Impact of Call-Site Optimizations on the Amount of Polymorphic and Megamorphic Calls - Full tables

Table 8. Full table displaying the impact of eliminating target duplicates on polymorphism: CD does not experience target duplicates, but the other benchmarks considered minimally-polymorphic see at least 50% of their polymorphic calls monomorphized.

	Number of calls		After eliminating target duplicates	
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**
BlogRails	956,515	63,319	-48.8%	-99.1%
ChunkyCanvas*	322	98	-80.0%	-100.0%
ChunkyColor*	320	98	-79.0%	-100.0%
ChunkyDec	322	98	-79.5%	-100.0%
ERubiRails	626,535	40,699	-37.4%	-98.6%
HexaPdfSmall	1,842,665	479,399	-21.7%	-99.6%
LiquidCartParse	821	280	-73.3%	-100.0%
LiquidCartRender	12,598	280	-84.1%	-100.0%
LiquidMiddleware	747	251	-68.8%	-100.0%
LiquidParseAll	5,369	280	-87.4%	-100.0%
LiquidRenderBibs	89,866	280	-73.7%	-100.0%
MailBench	81,886	12,697	-77.6%	-100.0%
PsdColor	14,053	233	-53.1%	-100.0%
PsdCompose*	14,053	233	-53.0%	-100.0%
PsdImage*	14,062	233	-53.0%	-100.0%
PsdUtil*	14,048	233	-53.0%	-100.0%
Sinatra	7,909	3,911	-82.8%	-94.4%
ADConvert	29,337	0	-58.3%	0.0%
ADLoadFile	22,654	0	-53.5%	0.0%
DeltaBlue	846	0	-33.7%	0.0%
PsychLoad	723,984	0	-85.7%	0.0%
RedBlack	8,718,802	0	-7.7%	0.0%
Acid	148	0	-81.1%	0.0%
BinaryTrees	148	0	-81.1%	0.0%
Bounce	149	0	-80.5%	0.0%
CD	4,638,337	0	-0.0%	0.0%
Fannkuch	148	0	-81.1%	0.0%
Havlak	1,344,909	0	-50.4%	0.0%
ImgDemoConv	149	0	-80.5%	0.0%
ImgDemoSobel	150	0	-80.0%	0.0%
Json	149	0	-80.5%	0.0%
List	148	0	-81.1%	0.0%
Mandelbrot	148	0	-81.1%	0.0%
MatrixMultiply	148	0	-81.1%	0.0%
NBody	148	0	-81.1%	0.0%
NeuralNet	190	0	-63.2%	0.0%
OptCarrot	3,477	0	-93.6%	0.0%
Permute	148	0	-81.1%	0.0%
Pidigits	148	0	-81.1%	0.0%
Queens	148	0	-81.1%	0.0%
Richards	148	0	-81.1%	0.0%
Sieve	148	0	-81.1%	0.0%
SpectralNorm	148	0	-81.1%	0.0%
Storage	149	0	-80.5%	0.0%
Towers	148	0	-81.1%	0.0%

Table 9. Full table displaying the impact of splitting on polymorphism: it fully addresses the remaining polymorphism in minimally-polymorphic benchmarks.

	Number of calls		After splitting		Number
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**	**of splits**
BlogRails	490,072	557	-100%	-100%	2163
ChunkyCanvas*	66	0	-100%	0%	43
ChunkyColor*	66	0	-100%	0%	42
ChunkyDec	66	0	-100%	0%	42
ERubiRails	391,997	553	-100%	-100%	1851
HexaPdfSmall	1,443,211	2,066	-100%	-100%	498
LiquidCartParse	219	0	-100%	0%	107
LiquidCartRender	2,000	0	-100%	0%	207
LiquidMiddleware	233	0	-100%	0%	114
LiquidParseAll	679	0	-100%	0%	136
LiquidRenderBibs	23,633	0	-100%	0%	191
MailBench	18,322	0	-100%	0%	343
PsdColor	6,586	0	-100%	0%	300
PsdCompose*	6,586	0	-100%	0%	300
PsdImage*	6,588	0	-100%	0%	300
PsdUtil*	6,584	0	-100%	0%	300
Sinatra	1,362	220	-100%	-100%	297
ADConvert	12,226	0	-100%	0%	236
ADLoadFile	10,525	0	-100%	0%	175
DeltaBlue	561	0	-100%	0%	78
PsychLoad	103,506	0	-100%	0%	78
RedBlack	8,043,472	0	-100%	0%	50
Acid	28	0	-100%	0%	27
BinaryTrees	28	0	-100%	0%	30
Bounce	29	0	-100%	0%	27
CD	4,638,217	0	-100%	0%	41
Fannkuch	28	0	-100%	0%	27
Havlak	666,933	0	-100%	0%	45
ImgDemoConv	29	0	-100%	0%	30
ImgDemoSobel	30	0	-100%	0%	27
Json	29	0	-100%	0%	27
List	28	0	-100%	0%	27
Mandelbrot	28	0	-100%	0%	27
MatrixMultiply	28	0	-100%	0%	31
NBody	28	0	-100%	0%	28
NeuralNet	70	0	-100%	0%	46
OptCarrot	221	0	-100%	0%	66
Permute	28	0	-100%	0%	28
Pidigits	28	0	-100%	0%	27
Queens	28	0	-100%	0%	28
Richards	28	0	-100%	0%	29
Sieve	28	0	-100%	0%	27
SpectralNorm	28	0	-100%	0%	30
Storage	29	0	-100%	0%	27
Towers	28	0	-100%	0%	27

B Impact of Call-Site Optimizations on the Amount of Polymorphic and Megamorphic Call-Sites - Full Tables

Table 10. Full table displaying the impact of eliminating target duplicates in the cache on the amount of polymorphic and megamorphic call-sites. This optimization is very effective and reduces at least 46.3% of polymorphic call-sites, up to 87.7%. Almost all megamorphic call-sites are eliminated, except in the four benchmarks with a higher amount of megamorphic call-sites.

	Number of call-sites		After eliminating target duplicates	
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**
BlogRails	1,015	210	-69.8%	-97.1%
ChunkyCanvas*	21	1	-86.0%	-100.0%
ChunkyColor*	21	1	-86.0%	-100.0%
ChunkyDec	21	1	-85.7%	-100.0%
ERubiRails	887	210	-70.2%	-97.1%
HexaPdfSmall	206	74	-78.6%	-98.6%
LiquidCartParse	53	5	-84.9%	-100.0%
LiquidCartRender	82	5	-80.5%	-100.0%
LiquidMiddleware	38	2	-71.1%	-100.0%
LiquidParseAll	65	5	-87.7%	-100.0%
LiquidRenderBibs	93	5	-76.3%	-100.0%
MailBench	163	34	-85.9%	-100.0%
PsdColor	119	7	-73.1%	-100.0%
PsdCompose*	119	7	-73.0%	-100.0%
PsdImage*	119	7	-73.0%	-100.0%
PsdUtil*	119	7	-73.0%	-100.0%
Sinatra	201	46	-86.6%	-95.7%
ADConvert	125	0	-80.0%	0.0%
ADLoadFile	96	0	-79.2%	0.0%
DeltaBlue	41	0	-46.3%	0.0%
PsychLoad	46	0	-87.0%	0.0%
RedBlack	51	0	-60.8%	0.0%
Acid	10	0	-80.0%	0.0%
BinaryTrees	10	0	-80.0%	0.0%
Bounce	10	0	-80.0%	0.0%
CD	13	0	-61.5%	0.0%
Fannkuch	10	0	-80.0%	0.0%
Havlak	12	0	-75.0%	0.0%
ImgDemoConv	10	0	-80.0%	0.0%
ImgDemoSobel	10	0	-80.0%	0.0%
Json	10	0	-80.0%	0.0%
List	10	0	-80.0%	0.0%
Mandelbrot	10	0	-80.0%	0.0%
MatrixMultiply	10	0	-80.0%	0.0%
NBody	10	0	-80.0%	0.0%
NeuralNet	11	0	-72.7%	0.0%
OptCarrot	26	0	-69.2%	0.0%
Permute	10	0	-80.0%	0.0%
Pidigits	10	0	-80.0%	0.0%
Queens	10	0	-80.0%	0.0%
Richards	10	0	-80.0%	0.0%
Sieve	10	0	-80.0%	0.0%
SpectralNorm	10	0	-80.0%	0.0%
Storage	10	0	-80.0%	0.0%
Towers	10	0	-80.0%	0.0%

Table 11. Splitting is very effective at reducing polymorphism: it fully addresses the remaining polymorphism and megamorphism, with the notable exception of our two Ruby-on-Rails benchmarks, where only a couple of polymorphic call-sites remain.

	Number of call-sites		After splitting		Number
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**	**of splits**
BlogRails	307	6	-99.3%	-100%	2163
ChunkyCanvas*	3	0	-100.0%	0%	43
ChunkyColor*	3	0	-100.0%	0%	42
ChunkyDec	3	0	-100.0%	0%	42
ERubiRails	264	6	-99.6%	-100%	1851
HexaPdfSmall	44	1	-100.0%	-100%	498
LiquidCartParse	8	0	-100.0%	0%	107
LiquidCartRender	16	0	-100.0%	0%	207
LiquidMiddleware	11	0	-100.0%	0%	114
LiquidParseAll	8	0	-100.0%	0%	136
LiquidRenderBibs	22	0	-100.0%	0%	191
MailBench	23	0	-100.0%	0%	343
PsdColor	32	0	-100.0%	0%	300
PsdCompose*	32	0	-100.0%	0%	300
PsdImage*	32	0	-100.0%	0%	300
PsdUtil*	32	0	-100.0%	0%	300
Sinatra	27	2	-100.0%	-100%	297
ADConvert	25	0	-100.0%	0%	236
ADLoadFile	20	0	-100.0%	0%	175
DeltaBlue	22	0	-100.0%	0%	78
PsychLoad	6	0	-100.0%	0%	78
RedBlack	20	0	-100.0%	0%	50
Acid	2	0	-100.0%	0%	27
BinaryTrees	2	0	-100.0%	0%	30
Bounce	2	0	-100.0%	0%	27
CD	5	0	-100.0%	0%	41
Fannkuch	2	0	-100.0%	0%	27
Havlak	3	0	-100.0%	0%	45
ImgDemoConv	2	0	-100.0%	0%	30
ImgDemoSobel	2	0	-100.0%	0%	27
Json	2	0	-100.0%	0%	27
List	2	0	-100.0%	0%	27
Mandelbrot	2	0	-100.0%	0%	27
MatrixMultiply	2	0	-100.0%	0%	31
NBody	2	0	-100.0%	0%	28
NeuralNet	3	0	-100.0%	0%	46
OptCarrot	8	0	-100.0%	0%	66
Permute	2	0	-100.0%	0%	28
Pidigits	2	0	-100.0%	0%	27
Queens	2	0	-100.0%	0%	28
Richards	2	0	-100.0%	0%	29
Sieve	2	0	-100.0%	0%	27
SpectralNorm	2	0	-100.0%	0%	30
Storage	2	0	-100.0%	0%	27
Towers	2	0	-100.0%	0%	27

References

[1] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing Dynamic Features in Python Programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 292–295. https://doi.org/10.1145/2597073.2597103

[2] Beatrice Åkerblom and Tobias Wrigstad. 2015. Measuring Polymorphism in Python Programs. *SIGPLAN Not.* 51, 2 (oct 2015), 114–128. https://doi.org/10.1145/2936313.2816717

[3] Michael Bächle and Paul Kirchberg. 2007. Ruby on Rails. *IEEE software* 24, 6 (2007), 105–108.

[4] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: A Basic Block Versioning JIT Compiler for CRuby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'21)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3486606.3486781

[5] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Nov. 2018), 108:1–108:30. https://doi.org/10.1145/3276478

[6] Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Ječmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 220 (nov 2020), 24 pages. https://doi.org/10.1145/3428288

[7] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.

[8] Alex Holkner and James Harland. 2009. Evaluating the Dynamic Behaviour of Python Applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91* (Wellington, New Zealand) *(ACSC '09)*. Australian Computer Society, Inc., AUS, 19–28. https://doi.org/10.5555/1862659.1862665

[9] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*. Springer, 21–38.

[10] Ross Ihaka and Robert Gentleman. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5, 3 (1996), 299–314.

[11] Thomas Kistler and Michael Franz. 2003. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems* 25, 4 (July 2003), 500–548. https://doi.org/10.1145/778559.778562

[12] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS'16)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[13] Yukio Matsumoto and Kiju Ishituka. 2002. Ruby programming language.

[14] Priya Nagpurkar. 2007. *Analysis, Detection, and Exploitation of Phase Behavior in Java Programs.* Ph. D. Dissertation. University of California Santa Barbara.

[15] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium.* IEEE, Rhodes Island, Greece, 10 pp. https://doi.org/10.1109/IPDPS.2006.1639325

[16] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1806596.1806598

[17] Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. 2013. Characteristics of Dynamic JVM Languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2542142.2542144

[18] Robert W. Scheifler. 1977. An Analysis of Inline Substitution for a Structured Programming Language. *Commun. ACM* 20, 9 (sep 1977), 647–654. https://doi.org/10.1145/359810.359830

[19] Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. 2017. TruffleRuby—A High Performance Implementation of the Ruby Programming Language.

[20] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI'17)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[21] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium* (Tucson, Arizona, USA) *(DLS'12)*. ACM, 73–82. https://doi.org/10.1145/2384577.2384587

Received 2022-07-01; accepted 2022-09-16