Context-Guided Splitting: Optimising Method Dispatch in Dynamic Languages

By

Sophie Kaleba

A thesis submitted to the University of Kent
in the subject of Computer Science
for the degree of Doctor of Philosophy

July 2024

Contents

Abstract

Dynamic languages are widely used, for instance on the server, in the browser, as well as for machine learning and scientific computing. Languages such as Ruby are attractive to developers, because they offer a diverse set of dynamic features, such as late binding, dynamic typing, and meta-programming capabilities. While these features grant a high degree of expressiveness, they also bring extra complexity to the runtime systems to achieve good performance. For instance, a common optimisation invented almost half a century ago minimises the run-time overhead of dynamic dispatch by using lookup caches. Splitting and inlining achieve similar goals. TruffleRuby is a modern runtime using these optimisations. It is built on top of GraalVM, a meta-compilation system that aims at re-using complex run-time components such as a just-in-time compiler between several language implementations.

In this dissertation, I investigate dynamic dispatch and method splitting in dynamic languages and aim to reduce their run-time overhead. As a first step, I investigate whether the long-held assumptions they use about call behaviour are still valid today and find that they indeed largely are. However, the analysis also uncovers that around 90% of splitting in TruffleRuby might be redundant. As a solution to this oversplitting, I propose *Context-Guided Splitting*, an alternative splitting heuristic that guides splitting decisions using contextual insight, i.e., the types of the current arguments of a call. It aims to reduce memory usage and compilation time by re-using previously split methods.

I evaluate the effectiveness of Context-Guided Splitting by comparing it with other splitting strategies on a diverse set of 37 Ruby benchmarks running on top of TruffleRuby, ranging from micro-benchmarks to larger, industry-inspired workloads. I show that Context-Guided Splitting successfully monomorphises 100% of megamorphic call-sites, and 99.3% (min. 96.8%, max. 100%) of polymorphic call-sites on average. Compared to the standard splitting approaches, it benefits polymorphic, and especially megamorphic benchmarks most, for which we observe a decrease of 56.6% of the time spent in garbage collection, and a decrease of run-time compilation time by 37.5% (min. -78.9%, max. 2.39×). Across the benchmark set as a whole, it decreases the AST size by 5.3% (min. -52.4%, max. 74.3%). With these results, we believe that Context-Guided Splitting will benefit large, real-world applications running on top of meta-compilation systems, making these systems more suitable for use in practice.

Acknowledgements

I would first like to thank my advisors, Stefan Marr and Richard Jones, for their invaluable help and support throughout the duration of my PhD. Thanks for your (seemingly) unlimited amount of patience. Thanks for your guidance and for sharing your expertise. Thanks for putting up with my time "management". I doubt I would have gotten to the end of it if it was not for your help. I would like to extend my thanks to the examiners, Jeremy Singer and David Castro-Perez, for their valuable advice and critical help to improve this manuscript. Likewise, I would like to thank Benoit Daloze for his help getting the hang of TruffleRuby and our discussions about splitting. I also want to thank Clément Boin for his feedback on an earlier version of this manuscript.

For having gotten me on track for this whole journey, I would like to thank Clément Béra. Not only did you introduce me to PLs and VMs –and research–, but you were also the first one to *subtly* suggest I could start a PhD. Thanks for your support. When I started getting into research, I was lucky to be advised by Gilles Grimaud, who subsequently advised most of my projects when I was at uni. Your kindness and benevolence surely guided my decision to embark on this journey.

I would also like to thank my friends from uni, Axel, Clément, Florian, Romain for all the fun times, and the nice synergy we had while working. Especially, I want to thank Célestine and Tamo, for our still ongoing daily chats and shenanigans.

While at Kent, I was lucky to have met some wonderful people. My hive mind: Octave, Caio and Calvin, Jonah and Max, Humphrey, Lisa, thanks for making Canterbury feel like home, and for spell-checking my broken English way too often. Of course, I am also grateful to my family for their constant support and encouragements. Merci pour tout.

Almost last, but definitely not least, I would like to thank my partner JB, whom I dragged out of his sunny Montpellier to live in Canterbury-the-grey. Thanks for putting up with me during all these years, and for supporting me all you could during this weird project.

Somebody wise once told me: “Sophie! You should be bragging about your work!”. And don’t get me wrong, I am sure they were right. However, I was lucky that, sometimes, I was not the one actually bragging about my work. I would like to thank Chris Seaton, for going the extra mile to advertise my research, and for his great help on this project. I’ll be forever grateful.

Chapter 1

Introduction

Many popular languages offer dynamic features, i.e., programming language mechanisms that act based on run-time information. Late binding (Moon 1974; Lewis, LaLiberte and Stallman 1990; Ungar and Smith 1987), when applied to method calls, is one of these features. It binds a method call to its target implementation at run time. This proves useful in the presence of polymorphic methods, i.e., when a method name is associated with several different method implementations, especially when combined with other dynamic features, e.g., dynamic library loading, or metaprogramming capabilities, such as intercession for field accesses for instance. These features are often considered to increase the expressiveness of the language, offering great freedom and flexibility to the developer. However, their implementation comes with engineering challenges (Ismail and Suh 2018), since they require extra effort to reach desirable performance.

Over the past decades, many dynamic optimisations have been introduced, notably to address these challenges. These optimisations are carried out at run time, while a program executes, since they rely on information available at run time. Inlining (Scheifler 1977) and lookup caches (Deutsch and Schiffman 1984) are examples of such optimisations, and are often used in combination to speed-up method calls. Many dynamic optimisations are built upon assumptions and common observations about how a program behaves at run time. For instance, lookup caches are built upon the observation that a given call-site in a program is likely to always end up activating the same small set of method implementations.

Dynamic optimisations have been investigated for decades, and are used in many dynamic language implementations today. Research on dynamic optimisations has emphasised execution time, and how to improve peak performance, i.e. the best possible performance that can be reached by a program. The now ubiquity of cloud computing, Software- and Function-as-a-Service, tends to shorten the duration of programs, all while imposing growing constraints on costs and memory. In such an environment, other run-time metrics have gradually gotten more interest from industry, such as fast start-up time and low memory footprint (Chevalier-Boisvert 2023; Menard 2023).

In the meantime, we also observe a growing interest from both research and industry on re-using complex components of run-time systems rather than developing solutions from scratch. This ideally cuts engineering costs, all while still giving access to a highly-performant system. Meta-compilation systems such as GraalVM (Würthinger et al. 2013) and RPython (Bolz and Tratt 2015) have been designed towards this goal: the language implementer only needs to provide an interpreter for their language, and benefits from the state-of-the-art run-time components offered by the meta-compilation system, such as the Just-In-Time (JIT) compiler or garbage collector.

1.1 Problem statement

Method calls often have a high performance overhead in dynamic languages, since many run-time operations and checks have to be performed for each method call to make sure the correct method implementation is executed. As a consequence, many dynamic optimisations focus on alleviating the cost of method calls. Lookup caches and inlining, which we mentioned earlier, are dynamic optimisations that may be used to improve the performance of method calls. Lookup caches hold the method implementations that have already been executed at a given call-site, saving on the cost of a potential lookup. Inlining replaces a call to a given method by the body of this method, potentially increasing the size of the compilation unit, while saving the cost of the method activation. The content of lookup caches often informs inlining decisions: if a lookup cache only holds a single method implementation (it is said to be monomorphic), then one can optimistically assume there will only be a single method implementation called at this given call-site, which means the call can be inlined. A lookup cache holding more than one method implementation is considered polymorphic, which will typically prevent inlining to be applied. Once a certain number of targets is added to the cache[1], the cache is considered megamorphic. Beyond this point, the lookup results are not cached any longer, which may lead to slow-downs.

The program representation used by meta-compilation systems embeds run-time information about the running program, such as types or shapes for instance (Würthinger et al. 2012). To reach peak performance, such representations benefit from aggressive monomorphisation. For calls specifically, it means that in addition to lookup caches and inlining, an extra dynamic optimisation called splitting (Hölzle, Chambers and Ungar 1991) may be applied. Sometimes, a single call-site may invoke more than one method, i.e., its lookup cache turns polymorphic, which may prevent the application of inlining or other optimisations. Lookup caches can be aggressively monomorphised through splitting: it does so by duplicating methods so that their caches and other specialisations start from a fresh state, e.g. ideally leading the caches to be monomorphic. This means that the program representation, usually bytecode or AST-based, will grow larger, since it contains methods and their duplicates, though these duplicates may ideally only contain

[1]This number is implementation-specific.

monomorphic caches. This uncovers optimisation opportunities that can be seized by the run-time system, to trigger inlining for instance. As a result, together with the other optimisations it helps to trigger, splitting helps optimising the execution by having a positive impact on various run-time metrics.

1.2 Research goals

In this thesis, I focus on the performance of method calls in a dynamic language implementation and investigate how to reduce their run-time overhead, especially regarding memory usage. More specifically, I aim to address the three following research questions:

Do the assumptions about behaviour used by common optimisations still apply in modern systems? [RQ. 1] Optimisations to reduce method call overhead, such as lookup caches, inlining and splitting, were designed decades ago, based on run-time behaviour observed back then. Since these optimisations are still used nowadays in more modern systems, I investigate whether these assumptions are still relevant.

Can we trigger fewer splitting operations, all while keeping the same level of call monomorphisation and can this improve run-time metrics, such as compilation time, start-up time and memory usage? [RQ. 2] Splitting has a positive impact on performance, but it relies on simple heuristics to be efficient and if triggered too often, may increase memory usage and compilation time, since more method clones are generated.

Can we re-use previous method clones resulting from splitting rather than triggering splitting? [RQ. 3] A single method may be split several times during execution, which means that several identical method clones may co-exist in the program's representation.

This thesis details the key concepts and outcomes of the research we carried out to address these questions.

Goal 1: Test commonly accepted assumptions about call-site behaviour. I first present the results of an analysis of the call-site behaviour of a diverse set of Ruby applications running on a modern language implementation. This aims at checking whether commonly accepted assumptions about the behaviour of programs still apply today, e.g., in more modern runtimes such as TruffleRuby that rely on meta-compilation, that we discuss in Section 2.3. [see RQ.1]

Goal 2: Use the abstract notion of Context to re-use method duplicates. I present Context-Guided Splitting, an alternative splitting heuristic that uses contextual insight to guide splitting decisions in a meta-compilation setting. With this technique, we aim to reduce compilation time and to decrease memory usage by dispatching to methods previously duplicated by splitting than triggering splitting itself, all while still monomorphising all calls when combined with other existing optimisations.
We use the notion of context to guide the splitting decision: conceptually, a context represents a set of run-time information that is available when a call is performed. In this work, we use the arguments' types, already available at a call-site to represent the context of a given call. Before deciding to split a method, we check whether a duplicate of this method exists for the same context. If this is the case, we dispatch to the previously generated method clone, rather than splitting the same method again. [see RQ.3]

Goal 3: Improve the memory footprint of splitting. By generating fewer compilation units, Context-Guided Splitting decreases the memory usage of a program, notably by reducing the size of the program representation. [see RQ.2]

Goal 4: Reduce the compilation time induced by splitting. Context-Guided Splitting reduces compilation time. Indeed, it dispatches to previously generated method clones, that are already specialised for a specific run-time typing context. If regular splitting had been triggered instead, a new compilation unit, i.e., the split method, would have needed to be optimised, then potentially compiled, increasing the time spent compiling. Therefore, by re-using method clones and reducing the number of splits, we reduce the total amount of compilation units that may need to be compiled, further decreasing the time spent in compilation. [see RQ.2]

Goal 5: Preserve the same level of call monomorphisation as standard splitting heuristics. Context-Guided Splitting avoids method duplication by dispatching to a method duplicate generated by a previous split. The dispatch happens if the context of the current call and the context in which the previous split was performed match. Sometimes, both contexts match, but lead to polymorphism in the shared method duplicate: this is what we call a context misprediction. If a context misprediction occurs, we fallback to using the standard splitting heuristics. Therefore, we do not expect a regression of the level of call monomorphisation since the standard splitting heuristics leads to a full monomorphisation of the program in the first place. [see RQ.2]

There are many topics closely related to this research, but which we consider to be out of the scope of this dissertation. Firstly, since monomorphic calls are not eligible for splitting, we do not aim to improve their performance with Context-Guided Splitting. We aim to leave their performance unchanged. In addition, we have not tested Context-Guided Splitting outside of a meta-compilation system and therefore can not claim similar results will be reached on another type of runtime[2]. Lastly, in Chapter 3, we us the content of lookup caches as a proxy for

[2]Though, Context-Guided Splitting is inspired by Flückiger et al. (2020)'s work, which is

a program's behaviour. While we show this approach may benefit to additional metrics, we do not study behaviour changes beyond the scope of lookup caches in this dissertation.

1.3 Contributions

The main contributions presented in this thesis can be summarised as follows:

- Identification of common assumptions used for optimising dynamic languages;
- Validation of these assumptions using large scale modern Ruby applications;
- Context-Guided Splitting, a new splitting heuristic guiding splitting decisions with contextual insight;
- Implementation of Context-Guided Splitting in TruffleRuby, a Ruby implementation built on top of GraalVM, a meta-compilation system;
- Study of the impact of Context-Guided Splitting on TruffleRuby performance and behaviour, compared to other available splitting strategies.

1.4 Key insights

We can use Context-Guided Splitting, a novel way of guiding splitting decisions with contextual insight, to limit the volume of splitting while still succeeding at monomorphising calls to the same extent as existing splitting strategies. Context-Guided Splitting, by re-using previously split methods, speeds-up compilation time and the time spent in garbage collection, as well as significantly reduces the size of the program representation.

implemented in Ř, a non-metacompiled runtime for the programming language R, and yields positive results in terms of performance.

The key insights of our study are presented in Chapters 3 and 5. They can be summarised as follows:

- Observing a diverse set of industry and research benchmarks written in a modern dynamic language implementation, most of the common assumptions about call-site behaviour are still valid. We observe that most calls are monomorphic, i.e., always have the same target. We also note that common call-site optimisations effectively monomorphise caches, and that industrial benchmarks tend to be larger in size, with bigger caches. However, we also find that call-site behaviour evolves during execution, and that the call-site behaviour between closures and methods differ, two assumptions that had been considered valid so far;
- Context-Guided Splitting reduces the size of the AST for megamorphic benchmarks best, on average by 25.3% (min. -52.4%, max. 74.3%), all while having a positive impact on the time spent in garbage collection by reducing it by 56.6% (min. -96.7%, max. 2.4%) and run-time compilation time, reducing it by 37.5% (min. -78.9%, max. 2.39×) for our megamorphic benchmarks;
- Context-Guided Splitting is as efficient as regular splitting in monomorphising the AST, while reducing the number of method clones generated;
- However, compared to regular splitting,Context-Guided Splitting tends to increase the execution time for benchmarks displaying low polymorphism by 23% (min. -23.4%, max. 65.73×) and polymorphic benchmarks by 18.6% (min. 10.6%, max. 56.6%) and increase the overall amount of allocated memory by 0.8% (min. -98.6%, max. 304×);
- A context only composed of argument types and identity of closure targets seems insufficient to confidently discriminate between call states.

1.5 List of publications

This dissertation builds directly and indirectly on the following papers I have contributed to:

[1] Kaleba, S., Larose, O., Jones, R. and Marr, S. (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*, Auckland, New Zealand: ACM Press, pp. 15–28

This peer-reviewed publication presents the common assumptions about a program's behaviour, and test whether they still apply today, through an analysis of a large set of Ruby benchmarks, running on top of an instrumented TruffleRuby system. This paper contributes to the contents of the Chapter 3 of this manuscript. Chapters 2 and 6 borrow from this publication as well.

[2] Larose, O., Kaleba, S., Burchell, H. and Marr, S. (2023). AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. New York, NY, USA: ACM Press

[3] Burchell, H., Larose, O., Kaleba, S. and Marr, S. (2023). Don't trust your profiler: An empirical study on the precision and accuracy of Java profilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, New York, NY, USA: ACM Press, MPLR 2023, pp. 100–113

[4] Aumayr, D., Marr, S., Kaleba, S., Gonzalez Boix, E. and Mössenböck, H. (2021). Capturing High-level Nondeterminism in Concurrent Programs for Practical Concurrency Model Agnostic Record & Replay. *The Art, Science, and Engineering of Programming*, 5(3), p. 39

I contributed to the writing and performance evaluation of these three publications. I reuse the same evaluation methodology in this dissertation, and use and enrich the scripts that were used in these publications to analyse and produce the plots presented in Chapters 4 and 5.

[5] Kaleba, S., Béra, C. and Miranda, E. (2018). Garbage collection evaluation infrastructure for the Cog VM. In *Proceedings of the 26th International Workshop on Smalltalk Technologies*, pp. 1–8

[6] Kaleba, S., Béra, C. and Ducasse, S. (2018). Assessing primitives performance on multi-stage execution. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pp. 1–10

[7] Kaleba, S., Béra, C., Bergel, A. and Ducasse, S. (2017). A detailed VM profiler for the Cog VM. In *Proceedings of the 25th International Workshop on Smalltalk Technologies*, pp. 1–8

These three peer-reviewed publications cover the different experiments I conducted on Squeak and Pharo, two modern implementations of Smalltalk. They do not contribute to specific chapters of this manuscript, but they support my research on highly performant run-time systems for dynamic languages.

Chapter 2

Common optimisations for dynamic languages

Dynamic languages such as JavaScript, PHP, Python, and Ruby are used to build a wide range of systems including, but not limited to, the back-end of web applications, scientific computing, and data-science and analysis. Most of these languages offer dynamic features like late binding, which we discuss in Section 2.1 of this chapter. While late binding provides great flexibility to the developer, it is often considered to add a run-time overhead compared to early binding, since extra checks need to be carried out at run time to ensure the correct method implementation is executed. In Section 2.2, we describe the most common optimisation techniques applied at run time, focusing on optimisations targeting calls (Section 2.2.2), and JIT compilation (Section 2.2.3).

Building a state-of-the-art JIT compiler requires a massive engineering effort to be effective. Section 2.3 introduces meta-compilation, a technique that aims at providing language developers with tools to reach the performance of a state-of-the-art language implementation with JIT compilation at a lower engineering cost, by re-using the complex parts of a run-time system between multiple language implementations. TruffleRuby is an example of such implementation: it uses GraalVM, a meta-compilation system, to implement the dynamic language Ruby. Since we use TruffleRuby in the remainder of this manuscript to build our experiments and run our analyses, we describe some of its specificities in Section 2.4.

2.1 The implementation challenges of late binding

Let's consider a simple program written in Ruby, depicted in Listing 2.1. It represents the implementation of the division operation, that converts the result to an integer. It is part of the `Numeric` class in the Ruby standard library.

Listing 2.1: The `div` method in the Ruby standard library, returning the result of a division rounded to the nearest integer.

```
1 def div(other)
2   raise ZeroDivisionError, 'divided by 0' if other == 0
3   (self / other).floor()
4 end
```

It contains a call to the method `floor()` on line 3. Binding is the process of pairing the method name, i.e. a symbol, in this example `floor`, to the correct implementation of the method `floor`. Ruby is a class-based, dynamically-typed language, and provides many dynamic features that make sound type inference unfeasible (see examples in Section 2.1.3). In that case, the "correct" implementation of `floor` is the one applying to the type of the result of the `(self / other)` operation.

Binding concerns many other programming languages' constructs beyond method calls, notably variables, but this section will exclusively focus on method calls. Binding may happen at compile time, or at run time (see Section 2.1.1). The latter enables the implementation of run-time polymorphism that, while offering many advantages, raises a number of implementation challenges, especially for languages featuring some form of reflection or dynamic code loading, since it complicates dynamic dispatch (see Section 2.1.3). Such features bring expressiveness and flexibility to the user, but may hamper performance, as discussed in Section 2.1.4.

2.1.1 Early and late method binding

Binding can generally be described as being early, as opposed to "late".[1] With early binding, the binding action is performed ahead of time, when the program is compiled, before being actually executed. With late binding, the binding action is realised at run time, when the program executes. It means that the call target, i.e. the bound method implementation, is not known before the call is executed. It may require, for instance, a run-time check in the form of a lookup to determine the dynamic type of the receiver of the method, and therefore the correct call target to execute. In practice, many dynamic languages' semantics require late binding. Late-binding is realised through the implementation of dynamic dispatch, which we describe in Section 2.1.3.

2.1.2 Example of a polymorphic method call

Previously, in Listing 2.1, we used the implementation of the integer division `div`, part of the Ruby standard library to illustrate what the term "binding" refers to. Line 3 of this snippet reads as:

```
(self / other).floor()
```

It contains two methods calls: one call to the `floor` method, denoted by the dot followed by the name of the method, and one call to a division between two objects, denoted `self/other`. Here, `self` and `other` represent the two operands of the

[1]The terms "static binding" and "dynamic binding" are often used interchangeably with early and late binding.

`div` operation. If the `self/other` division succeeds, then the result is floored, i.e., rounded down to the nearest integer.

The left operand of the `self/other` operation, `self`, is of type `Numeric`, since `div` is implemented in the `Numeric` class.[2] The right operand, `other`, can potentially be of any type since Ruby is dynamically typed. Several implementations of the method `floor` exist in Ruby, and the implementation that will be executed depends on whether the result of the `self/other` division is a rational number, a float or an integer. The method `floor` is therefore said to be "polymorphic", i.e. it represents a single name to different method implementations.

Polymorphism allows for great language expressiveness but requires to disambiguate which method implementation is going to be executed at a given call-site.

2.1.3 Dispatching to the correct method implementation, at run time

Why defer until run time? With Listing 2.1, we showed that polymorphism requires language implementations to disambiguate which method implementation should be invoked at a given call-site. In a language like Ruby, this task may quickly turn arduous. Indeed, since Ruby is dynamically typed, it means that in our example the call to `floor` might not succeed if `other` is of an non-applicable type, e.g. a string. Since the type is only checked at run time, it means that the identification of the correct method implementation to invoke must be deferred until run time, too. That is the reason why this call is late bound.

Beyond dynamic typing, many language implementations feature dynamic library loading, or metaprogramming capabilities, such as reflection or dynamic code execution through the use of `eval` for instance. With such features, a class's structure can be modified at run time, with the addition of new methods, or new

[2]`self` in Ruby corresponds to the current instance of the class. It is the equivalent to `this` in Java for instance.

classes in the hierarchy. In Ruby, such dynamic changes to classes are colloquially called monkey-patching. In JavaScript, for instance, it is possible to add new methods at run time by modifying the prototype of an object. Such features further enforce the need for late binding.

Late binding is realised through the implementation of dynamic dispatch, which is the process of dispatching to the correct method implementation at run time.

Method lookup is the core mechanism behind dynamic dispatch. Broadly speaking, lookup uses the current state of the call environment, e.g., the run-time types of arguments, the run-time type of the receiver or the currently active module, to identify the method implementation that needs to be invoked at a given call-site. Lookup can take many forms: in the case of an object-oriented, class-based language for instance, as in our example in Listing 2.1, the call to `floor()` on line 3 is resolved by looking an implementation of `floor` in the class hierarchy, starting from the current run-time class of the result of the division `self/other`. If no method with the name `floor` is found in the starting class, lookup continues upwards in the super class(es)[3]. If a language features modules, it may be necessary to traverse the module hierarchy to find an implementation of the function.

Once the correct method implementation has been identified via lookup, it can then be activated. Activation is the act of passing the parameters and setting up the stack frame, to then starting the execution of the looked-up method, i.e. executing each statement of the body of the method.

Single or multiple dispatch? Method dispatch can be said to be “single”, as opposed to “multiple” dispatch. Languages featuring single method dispatch will perform the method lookup based on the type of the first argument of the method,

[3]In Ruby, everything is an object, and numbers are represented by instances of the classes `Float`, `Rational`, or `Integer`.

usually the receiver of the call. This is the case in our example in Listing 2.1. With multiple dispatch, the lookup is based on more data, i.e., the run-time type of all the arguments for instance, as in, e.g. Lisp (Moon 1974), or Julia (Bezanson et al. 2015).

2.1.4 The impact of late-binding on performance

The late binding of method calls offers flexibility to both developers and language implementers. It has been popularised in earlier implementations of Lisp (Moon 1974; Lewis, LaLiberte and Stallman 1990) and Self (Ungar and Smith 1987). Late binding is still part of the semantics of many popular languages used nowadays, such as Java, C++, Python, Ruby or R for instance.

Performance-wise, in comparison to early binding, late binding adds an extra run-time overhead (Zendra and Driesen 2002; Pontelli, Ranjan and Gupta 1998; Milton, Schmidt et al. 1994), since method lookup is performed at run time. Several projects have been investigating how to provide an ahead of time (AOT) implementation of languages having dynamic features, therefore ideally avoiding late binding. For instance, Hopc (Serrano 2018), and SJS (Choi et al. 2015) are both AOT JavaScript compilers, though SJS only apply to a subset of JavaScript. Similar experiments have been conducted for Scheme (Yvon and Feeley 2021) or Lisp, despite both languages featuring many meta-programming capabilities. Java has also been AOT compiled, see notably the native image project (Wimmer et al. 2019), part of GraalVM ecosystem. To be applicable, the latter operates on closed-world assumptions, that limit the use dynamic features such as, e.g. reflection. Many of these AOT compilers also usually apply to only a subset of these languages, limiting expressiveness. Alternatively, many other works have historically investigated how to optimise the performance of languages that feature late binding, at run time. We describe such approaches in the next section.

2.2 Improving performance, at run time

Many different strategies have been devised in the last decades to improve the performance of dynamic languages, notably with the design of dynamic optimisations, a kind of optimisation carried out during execution and that depends on run-time information.

Calls are often targeted by optimisations, especially once the run-time overhead induced by late binding comes into the mix. Many dynamic optimisations have focused on improving the performance of calls. We describe some of the most common call optimisations, e.g., lookup caches, splitting and inlining, in Section 2.2.2. We then explain in Section 2.2.3 how such optimisations may fit within a multi-tier execution run-time system. These optimisations may sometimes rely on speculations about the run-time behaviour of the program to generate efficient code, usually as part of the last execution tiers. We explain what speculative optimisations entail in Section 2.2.4.

2.2.1 Code representation

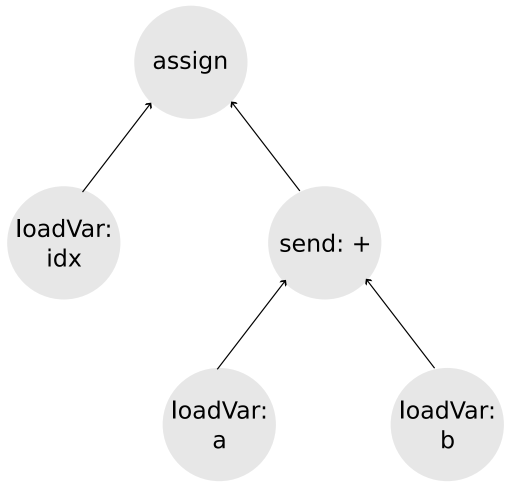
Programs are internally represented in a form different from plain source code as it typically eases the future processing and helps applying optimisations. Two representations are commonly used by interpreters: Bytecodes and Abstract Syntax Trees (AST).

Bytecodes act as an abstract instruction set for either virtual stack or register machines. These operations usually include `push`, `pop`, `send` and `jump` operations (stack-based, as seen in Figure 1a) or `move`, `load`, `call`, `goto` operations (register-based). Bytecode opcodes are in general a few bytes long at most, which make this representation memory-efficient. However, bytecodes make it complex to infer the control-flow structure of the application, due to their abstract nature.

ASTs are frequently used as an alternative to bytecode to represent code. The

```
POP a_var
POP b_var
SEND add_fun
PUSH idx_var
```

(a) A stack-based, bytecode representation



(b) An AST representation

Figure 1: The bytecode and AST representations for a simple `idx = a + b` statement. Bytecodes and AST are the two most common code representations used for interpretation.

code is represented as a tree (see Figure 1b), usually a compact version of the parse tree built by the parser, which makes the AST an easy-to-build and easy-to-read representation, though heavy on memory.

Run-time systems often combine several execution tiers (see Section 2.2.3). The first tier(s) usually rely on an interpreter, while the later tiers rely on compilers to execute code, and each of them may rely on several forms of internal representations (IR). ASTs and bytecodes are often used as representations when interpreting dynamic languages. For JIT compilation, compilation units are typically first converted to representations based on Sea-of-Nodes or Single-Static-Assignment (SSA) form. Sea-of-Nodes, which is an extended graph, are especially useful for, e.g., global value numbering or constant propagation (Click and Cooper 1995; Click and Paleczny 1995). It is notably used in Java/Hotspot. This representation inspired GraalVM's IR (Duboscq et al. 2013), which is a graph combining data-flow and control-flow edges. The SSA form (Cytron et al. 1991) is also a very common form of IR that enforces variables are only used once, which is useful for, e.g., register allocation. It is notably used in V8 as part of the Maglev JIT compiler[4] and in the LLVM IR (Lattner and Adve 2004). Abstract assembly is another code representation that has been used in, e.g., JikesRVM (Alpern et al. 2005).

2.2.2 How to optimise late-bound calls?

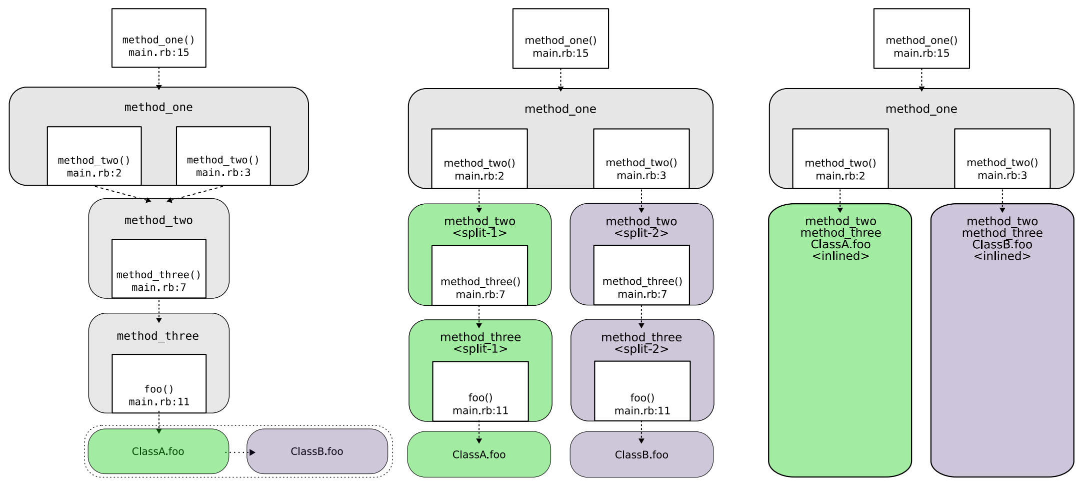
Method calls are the object of various run-time optimisations. Lookup caches, used together with method splitting and inlining, typically help to optimise away the performance costs of calls. Used in combination with other optimisations, they help producing highly efficient code.

Lookup caches. As first described by Deutsch and Schiffman (1984), lookup caches are associated with call-sites and aim to avoid repeated and possibly time-consuming dynamic method lookups. Lookup caches typically hold the call targets observed at a call-site, along with the criterion of when the given target is valid. 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 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 optimisation.

A cache turns polymorphic when it holds more than one call-target. We represent a lookup cache turning polymorphic in Figure 2a. The grey rectangles

[4]See `https://v8.dev/blog/maglev`. Last consulted 25/02/2025.



(a) Lookup caches (b) Splitting (c) Inlining

Figure 2: The lookup cache of the `foo()` call-site in Figure 2a is polymorphic with two entries. Splitting can be used to de-duplicate call-sites and monomorphise the call tree, as seen in Figure 2b. It is therefore easier to inline the monomorphised branches of the AST to get rid of the calls to `method_three` and `foo` through inlining (Figure 2c).

with rounded corners represent methods, i.e., call-targets. The white rectangles represent call-sites. In this figure, the call-site of `foo`, at the bottom of the AST, has seen two different call-targets, one for `ClassA` and one for `ClassB`. The lookup cache is therefore polymorphic with two entries, contained within the dotted line. When the maximum number of call-targets is exceeded, this maximum being system-dependent, the cache is deemed megamorphic, which in some systems means that there is no further caching at this call-site, since too many *(Receiver type, Call-target)* tuples make caching ineffective.

Splitting. First described by Chambers and Ungar (1989), also called method cloning, splitting duplicates a method in the context of the caller. The new copy starts with an empty lookup cache to monomorphise potentially polymorphic call-sites. It is similar to inlining (see below), but does not embed the code in the calling method and thus, keeps the two compilation units separate. In Figure 2b, the `method_two` and `method_three` methods have been split: they exist in two variations in the AST, denoted by the `<split-n>` notation and their different colours. After splitting, the two `foo` call-sites have monomorphic lookup caches.

Inlining. Inlining (Scheifler 1977) is an optimisation 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 JIT compiler, enables further compiler optimisations 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. However, this is not always the case; the meta-compilation system GraalVM, that we discuss in Section 2.3, may inline polymorphic call-sites if their size does not exceed a specific threshold. A potential outcome of inlining is depicted in Figure 2c: since the caches have been monomorphised by splitting, the calls to `method_three` and `foo` can be inlined into the `method_two` call-target, to form a single compilation unit.

2.2.3 Just-In-Time compilation

These dynamic optimisations take place in run-time systems featuring JIT compilation, which is a dynamic optimisation technique where code is compiled on the fly, during execution, rather than ahead of time. JIT compilation aims at improving the run-time performance of an application by taking advantage of information only available at run time via profiling to produce highly optimised code.

Multi-tiered execution. Interpreters are usually the first favoured choice when it comes to language implementation: they are relatively easy to implement, with a fast execution cycle, that makes them easier to debug and maintain (Ertl and Gregg 2003; Romer et al. 1996). Usually, the code is parsed into some representation that is then interpreted. The two most common code representations are AST and bytecodes, as seen in Section 2.2.1. However, it is often considered that naive interpreters are slow: notably, a repeated sequence of code is likely to be interpreted several times (Ertl and Gregg 2003).

To cope with these issues, some approaches focus on implementing more performant interpreters, for instance by using quickening (Brunthaler 2010), super-instructions (Piumarta and Riccardi 1998) and supernodes (Larose, Kaleba and Marr 2022; Basso, Bonetta and Binder 2023), or self-optimising interpreters (Würthinger et al. 2012). However, most Virtual Machines (VM), i.e., run-time systems, rather rely on both interpretation and JIT compilation.

The first runs of the code are usually interpreted, and the execution is then transferred to a baseline JIT compiler–the second execution tier–, that will translate high-level code into native code. During these stages, the system profiles the program to gather run-time information about its behaviour, such as the executed methods and how many times they have been called, the types of objects or the branches taken for instance. Most VMs now rely on selective compilation (Schilling 2003): a subset of code is selected according to a specific heuristic (see Section 2.2.4) and is then compiled, improving the memory footprint of the application. Usually code compiled in the second execution tier is subject to none or only basic optimisations, such as simple inlining.

High-performance VMs have in general three execution tiers with a third one in which an optimising compiler performs more aggressive optimisations (Hartmann, Noll and Gross 2014). The compiled code is thus highly performant, though it implies the system must depend on elaborate deoptimisation strategies in case a speculation fails (Zheng, Bulej and Binder 2017). In JavaScriptCore, the third execution tier is meant to produce optimised code with low latency, while an extra fourth tier compiles high-throughput code (Tadeu Zagallo 2019). We represent it in Figure 3.

Java HotSpot has five execution tiers: the second to fourth tiers rely on the C1 compiler, with an increasing level of profiling and instrumentation. The fifth

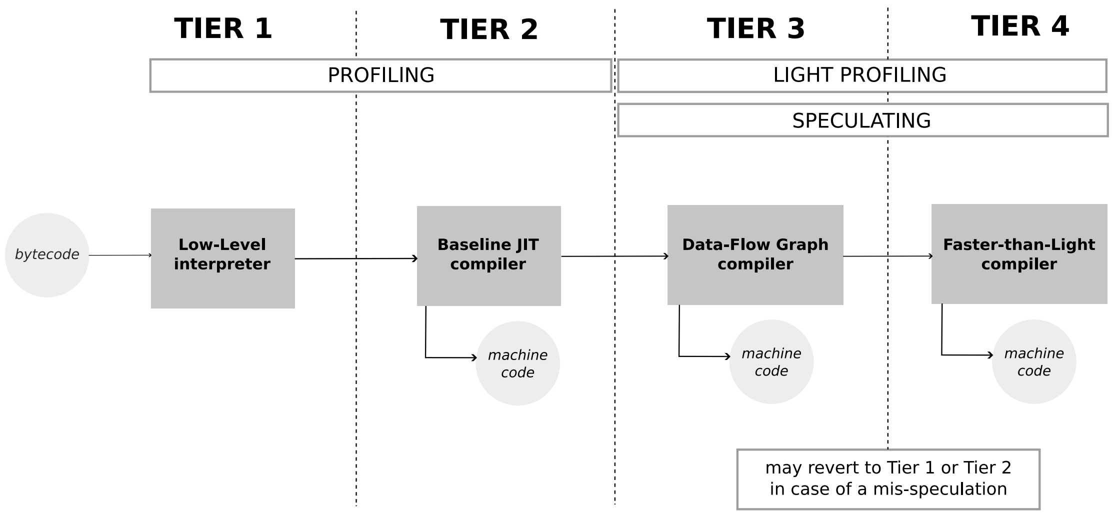


Figure 3: The JavaScriptCore engine has four execution tiers. The lower tiers are used to gather profiling information that will be used in the following tiers to produce highly efficient code.

tier relies exclusively on the optimising compiler C2 (Evans, Gough and Newland 2018).

Compilation unit. In the context of multi-tiered execution, one challenge is to determine which part of the code should be compiled. Indeed, the chosen strategy is likely to affect both the quality and size of compiled code, as well as the overall overhead of profiling. Methods have been traditionally used as compilation units, pioneered by SELF (Chambers, Ungar and Lee 1989). However, sometimes only part of a method is actually “hot”, i.e. frequently executed.[5] In response, execution traces (i.e. instructions sequences) have been alternatively used as compilation units to record the most commonly taken paths (Gal et al. 2009). Bala, Duesterwald and Banerjia (2000) identify hot traces to be optimised in their Dynamo VM, by monitoring and counting specific trace points, notably backward taken branches (i.e. loops). The PyPy project also relies on traces, but at the interpreter level rather than the application level (Bolz et al. 2009). This

[5]Hot spots are compilation units that are frequently executed and can therefore be considered for compilation.

approach is further detailed in Section 2.3. Basic blocks have been also used as units, first by Hansen (1974) and more recently by Chevalier-Boisvert et al. (2021), while region-based compilers use regions, which comprise of a set of basic blocks (Hank, Hwu and Rau (1995), Ottoni (2018)).

2.2.4 Speculating about behaviour and optimisation heuristics

Speculating for performance. Most dynamic optimisations require information that can not be known ahead of time. For instance, dynamic code loading may modify the structure of a class at run time, which means that a new method may be added in a given class when the program runs. Such a use-case makes it challenging to assume that a given call-site will always be monomorphic for instance, since a new implementation of the method might be added in the future, which may in turn prevents inlining to be applied.

Therefore, a language implementer may want to speculate about the state of the program, or specific values. In that case, by speculating that the call-site will stay monomorphic with the implementation activated so far, so it could be inlined. These speculations are used to apply optimisations optimistically, assuming that they are correctly predicting the future behaviour of the running program, potentially leading to better performance.

Optimisation heuristics. For many dynamic optimisations, one of the main challenge is to determine the right time to apply them so they are beneficial for performance. For instance, JIT compilers both profile and optimise at run time. This induces an overhead that has to be outweighed by the benefits obtained from the optimised compiled code. Often the costs come into consideration to guide the optimisation strategy; Kulkarni (2011) studied if and when they should compile methods to get the best benefits; Brock et al. (2018) describe their cost-based policy for their JIT compiler. They compute the cost/benefit of compiling a specific method at a given time based on memory space and processor time. They use the result to determine how many times a method should be interpreted before getting compiled. In the context of multi-tiered execution, one of the main challenges is to determine the right time to trigger compilation and move on to the next tier.

Common assumptions about run-time behaviour. A lot of dynamic optimisations are built upon widely accepted assumptions about the run-time behaviour of a program. For instance, most compilation heuristics are built upon the assumption that a small part of the application code accounts for most of the time spent in this application, notably the hot spot detection approach. Knuth's observation (Knuth 1971) that frequently executed sequences of code should still be frequently executed in the future justifies that they should be compiled (Lee, Moon and Kim (2008), Chambers, Ungar and Lee (1989)). This strategy was first implemented by Hansen (1974) where basic-blocks were compiled once they reached a predetermined execution counter threshold. It has since been widely adopted (see JavaScript's V8 (Google 2008) or Jikes RVM (Alpern and Attanasio 1999)), usually in more advanced forms, relying on counters (Paleczny, Vick and Click 2001), or sampling (Whaley 2000). Some other heuristics have been investigated in combination with the hot spot strategy, such as the size of methods eligible for compilation (Schilling 2003), or the presence of loops (Suganuma et al. 2005).

Besides JIT compilation, implementing lookup caches assumes for instance that the behaviour at a given call-site is likely to stay stable and that only a small set of different receivers will be encountered, since the more polymorphic a lookup cache is, the less efficient it becomes. Splitting relies on a similar reasoning, since ideally the cloned method should only be specialised towards one type. We further discuss these common assumptions in Chapter 3, and check whether they still hold nowadays.

2.3 Re-using parts of run-time systems via meta-compilation

Dynamic languages rely on runtime components such as JIT compilers and their optimisations to achieve satisfactory run-time performance. However, high performance JIT compilers are complex systems that require a lot of engineering effort to achieve satisfactory results; meta-compilation techniques have been aiming to re-use many complex parts of the run-time system, such as the JIT compiler or the garbage collector, between many different languages implementations. They do so by producing language-agnostic intermediate representations that can then be passed on to these components and optimised so that highly efficient machine-code is generated, all while requiring the language implementer to only provide an interpreter.

The two main meta-compilation techniques are meta-tracing and partial evaluation.

Meta-tracing. The RPython system (Bolz et al. (2009), Bolz and Tratt (2015)) creates a tracing-JIT for a target language. In this case, when a hot loop is detected, the interpreter running the user program gets traced, rather than the executed user program, as it is the case in standard, i.e., non-meta, tracing JITs.

Partial evaluation is another meta-compilation approach, adopted by GraalVM (Wimmer and Würthinger 2012), and inspired by Futamura (1983), that we will detail more in depth. Rather than relying on a trace from interpreter levels, this approach starts with the self-optimisation of the application AST guided by run-time type information (Würthinger et al. 2012). With this approach, an AST node representing an addition may get specialised to only represent an addition between two integers for instance, if the two operands have been profiled as integers.

Once the AST is deemed stable, which means the specialisations that have been applied do not change any more, the partial evaluator compiles the specialised AST, i.e., the profiled data together with the relevant code part of the AST nodes, to produce a language agnostic IR that will be optimised by the Graal compiler (Würthinger et al. 2017).

In this thesis, we aim at analysing and optimising modern run-time systems, such as a meta-compilation system. Since we require our analysis to be conducted on a diverse enough benchmark set, we choose TruffleRuby as our experimental platform in the remainder of the manuscript. TruffleRuby is an implementation of the Ruby programming language on top of the meta-compilation system GraalVM, therefore using the Partial Evaluation approach.

2.4 A focus on TruffleRuby

We instrumented TruffleRuby to monitor the behaviour of Ruby applications (see Chapter 3), and implemented an optimisation for splitting, called Context-Guided Splitting, as part of the TruffleRuby runtime (see Chapter 4). We present the results of this experiment in Chapter 5. In this section, we describe the implementation-specific details of TruffleRuby that are relevant to our experiments, as well as some aspects of Ruby semantics.

2.4.1 The GraalVM execution tiers

TruffleRuby has multiple execution tiers, similarly to what we discussed in Section 2.2.3. We show how these execution tiers interact with one another in Figure 4.

In the first tier, the execution of Ruby code is handled by an AST interpreter. The AST gets specialised according to the run-time types observed, and is afterwards partially-evaluated and compiled to the Graal IR. The Graal JIT compiler, used in both tiers two and three, then optimises this IR to produce efficient machine

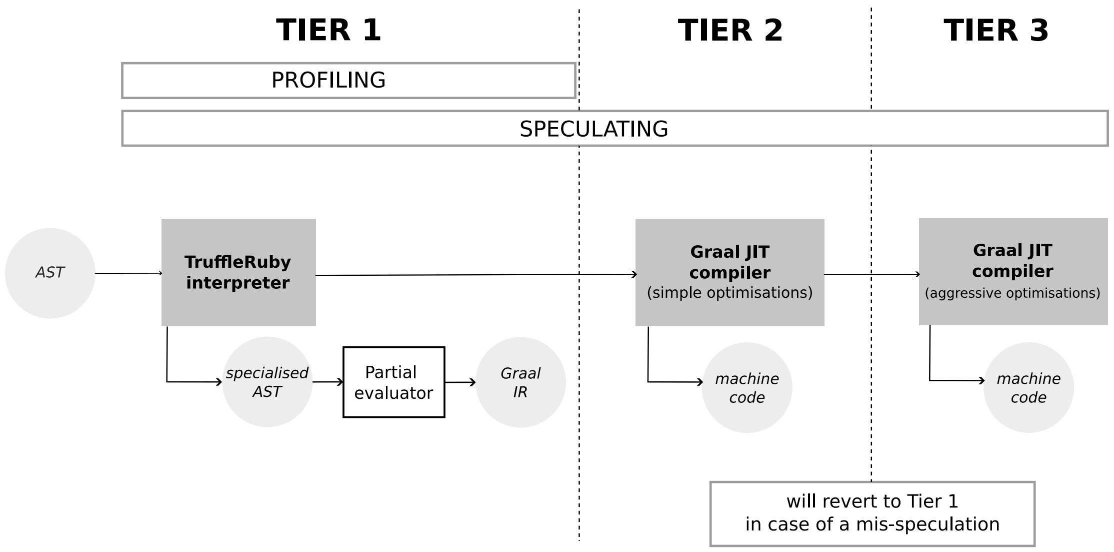

TIER 1
TIER 2
TIER 3
PROFILING
SPECULATING
AST
TruffleRuby interpreter
specialised AST
Partial evaluator
Graal IR
Graal JIT compiler (simple optimisations)
machine code
Graal JIT compiler (aggressive optimisations)
machine code
will revert to Tier 1 in case of a mis-speculation

Figure 4: TruffleRuby has three execution tiers, with the last tier aiming at better peak performance. The same JIT compiler is used in Tier 2 and 3.

code. Fewer optimisations are performed during Tier 2 compared to Tier 3, and these optimisations are generally more lightweight. The goal of Tier 2 is to escape from the slower, interpreted mode earlier, to improve startup performance. Tier 3 will further optimise the IR to produce higher quality code and reach better peak performance. For instance, only a subset of hot methods is considered for inlining during Tier 2, but this subset grows larger in Tier 3. Splitting, as well as Context-Guided Splitting, both happen in Tier 1.

2.4.2 Methods and Closures in Ruby

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

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`. TruffleRuby will inline many of these builtins at the AST-level to avoid the call overhead, and allow specialisation 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 number of arguments more strictly than procs. Procs are more forgiving of mismatching arguments. Block literals in the code instantiate as procs. In this dissertation, we will not distinguish these flavours further.

2.4.3 Optimisations in TruffleRuby

Lookup caches, splitting, and inlining are important optimisations used in dynamic language implementations, as mentioned in Section 2.2.2. TruffleRuby relies on these optimisations for its interpreted and JIT compiled performance. While lookup caches minimise lookup overhead and profile call-site behaviour, splitting and inlining reduce the number of call-targets at a call-site to improve performance by reducing the polymorphism.

Lookup caches. 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.

Inlining. 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 2.4.2).

Splitting. In TruffleRuby, splitting is done at the method level and may be triggered when a method with a polymorphic lookup cache is called. It copies an uninitialised 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 uninitialised 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 5,[6] `foo()` on line 11 of Figure 5c becomes polymorphic after being indirectly called on a ClassA and ClassB from lines 2 and 3. 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 GraalVM). In this example, `method_two` and `method_three` are marked for splitting so that the call-sites on lines 2 and 3 will get separate, specialised copies. Figure 6 shows the impact of this splitting action on a simplified AST structure of this program: splitting has effectively cloned `method_two` and `method_three` so the AST now contains four fresh nodes that will ideally stay specialised against `ClassA` (green)

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

(a) Implementation of foo in ClassA

```ruby
def foo()
  a = Time.now()
  puts a+"-A"
end
```

(b) Implementation of foo in ClassB

```ruby
def foo()
  a = Time.now()
  puts a+"-B"
end
```

(c) Main class

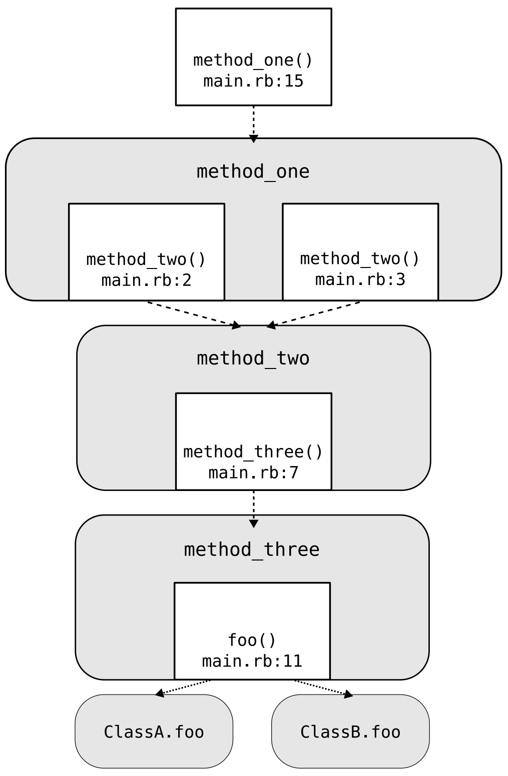
```ruby
def method_one()
  method_two(ClassA.new)
  method_two(ClassB.new)
end

def method_two(arg)
  method_three(arg)
end

def method_three(arg)
  arg.foo()
end

for i in 1..3 do
  method_one()
end
```

Figure 5: This small Ruby program triggers the splitting of the `method_two` and `method_three` methods. The amount of splitting could be avoided if it was possible to dispatch on previously split methods.

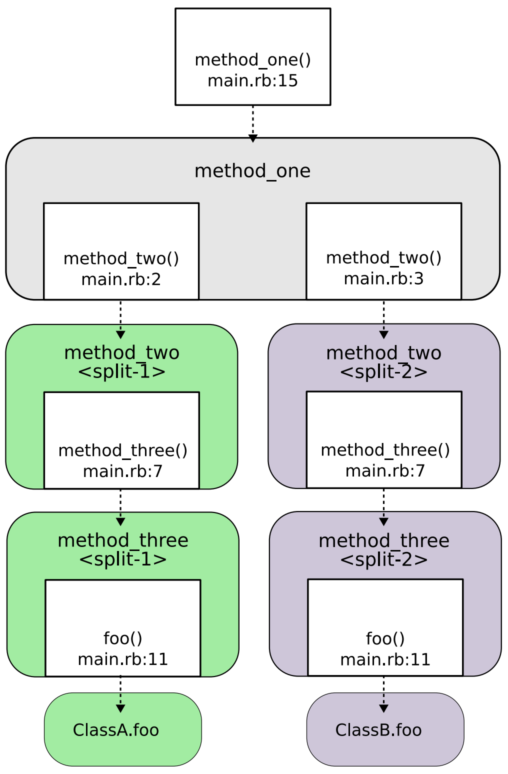
and `ClassB` (purple).

Additionally, TruffleRuby will force splitting for some known methods in the core Ruby library, so that they specialise in the context where they are used (see Section 2.4.3).

Eliminating Call Target Duplication Classic lookup caches contain entries of *(Receiver type, call-target)* (see Section 2.2.2). In Truffle, method calls are realised with the framework's call node (called `DirectCallNode`), which points to the call-target and implements splitting and triggering of the JIT compiler. So, in GraalVM-based languages, a lookup cache would contain a *(Receiver type, call*



(a) AST of program from Figure 5 before splitting



(b) AST of program from Figure 5 after splitting

Figure 6: Splitting decouples the application's structure so it contains new branches that are specialised against a specific type, here ClassA (tree in green) and ClassB (tree in purple).

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_nums = [3.45983662, 189]

for e in list_of_nums do
    e.div(3)
end
```

The two elements of `list_of_nums` are both numbers, but the first is an instance of the class `Float` and the second an `Integer`. They share the same `div` implementation, i.e. the same target implementation of `div`, which we already saw in Listing 2.1, earlier in Section 2.1.2. 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.

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.[7] Thus, all entries in the lookup cache at an application site contain call targets that can fully specialise 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.

[7]After discussing our results with the TruffleRuby team, this has been changed, and closure calls rely on Truffle's normal splitting.

Chapter 3

Do the common assumptions about run-time behaviour still hold?

In the previous Chapter, I set up a list of assumptions about the behaviour of programs that were used to build the common JIT optimisations that are still in use nowadays, especially assumptions about the behaviour at call-sites, for instance, the fact that most call-sites only have one type of receiver (i.e., monomorphic call-sites).

From Section 3.2, I analyse the behaviour of a diverse set of benchmarks written in Ruby, comprising of micro-benchmarks and larger, industrial benchmarks, and check whether the assumptions about a program's behaviour still apply today, in the context of meta-compilation. I especially focus on the content of lookup caches and how it is influenced by existing optimisations, notably splitting (Section 3.2.2). I show that splitting, despite its effectiveness in monomorphising lookup caches, is triggered too often (Section 3.2.3). I show that the most common assumptions about call-site behaviour still hold in the context of meta-compilation, with the exception of the stability of the cache state, detailed in Section 3.2.5.

3.1 Checking the assumptions about behaviour in today's context, in TruffleRuby

We mentioned in Section 2.2.4 that many dynamic optimisations and heuristics rely on commonly accepted assumptions about a program's behaviour. In this Chapter, we want to check whether these assumptions still hold today by analysing the call-site behaviour of applications running on TruffleRuby. This Section details our benchmark set and describes the methodology used for this analysis.

3.1.1 Motivation and Terminology

Analysing call-site behaviour. Coding practices often encourage small methods and code reuse in various forms. Together with paradigms such as everything is an object (Goldberg and Robson 1983; Matsumoto and Ishituka 2002) or everything is a call (Ihaka and Gentleman 1996), method or function calls often end up dominating an application's behaviour. Having many calls to many small methods can be expensive if calls are not optimised. Fortunately, widely adopted optimisations such as inline caches (Hölzle, Chambers and Ungar 1991) minimise lookup overhead by caching the target method, and inlining, which replaces a method call by the method's implementation in the caller, avoids the call overhead and enables further optimisations.

These optimisations rely on the assumption that the behaviour of an application stabilises 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 behaviour (Perelman et al. 2006; Nagpurkar 2007). 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 (Hansson 2004), or Instagram with millions of lines of Python code, we are interested in verifying whether these assumptions are still likely to hold.

Terminology. In our analysis, we distinguish call-sites and call-targets to characterise the call-site behaviour 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 behaviour** to characterise the state of a call-site at run time. In this paper, we use its lookup cache state (see Section 2.4.3) and content (i.e. the run-time types of its receivers) as a proxy for its behaviour. We use this information to characterise a call-site and observe how it evolves at run time.

3.1.2 Selection of the benchmark set

We use benchmarks from the Are We Fast Yet project (Marr, Daloze and Mössenböck 2016), which contains nine microbenchmarks and five 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

(Chevalier-Boisvert et al. 2021), which are production projects funded by Oracle and Shopify respectively. These are two industry-driven Ruby implementations, where the engineering teams each maintain their own benchmark suites with workloads relevant for their corresponding key customers. Compared to the Are We Fast Yet benchmarks, these benchmarks tend to be larger, aiming to guide the development of the two corresponding Ruby implementations. YJIT's benchmarks are of particular interest since they are meant to resemble the real-world workloads of Shopify.[1] These benchmarks include web server applications such as ERubiRails and BlogsRails, which are built on top of the large Ruby-on-Rails framework (Hansson 2004) and rely on template processing. Since Ruby is often used for such web applications, they represent a major use case and optimisation 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, like the Liquid benchmarks that use the Shopify's Liquid templating language, to render a shopping cart's content for instance. The full list of benchmarks is available in Appendix A.1, with a brief description of each benchmark.

3.1.3 Behaviour monitoring and analysis

We instrumented the TruffleRuby runtime 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 3.2.7.

Trace description. Figures 7–9 show excerpts of an execution trace for the DeltaBlue benchmark, from the Are We Fast Yet set of benchmarks. Each line of the trace represents one call. The first number is the position of the line in the trace, for instance `493` on the first line of Figure 7, which means that this call

[1]Shopify funds YJIT's development.

is on the 493rd line of the trace. We use this information to distinguish between calls, and to sort them chronologically. 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 2.4.3 for a definition, and Figure 8 for such an example in 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. 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 analyse these traces, compute statistics, and generate summary plots and tables.[2] The results of the analysis are described in detail in Section 3.2.

Behaviour reconstruction. We use the trace to assess the state of the call-sites' behaviours and to identify the impact of call-site optimisations. The lookup cache state is reconstructed by counting the number of different receivers at a given call-site. For example, in Figure 7 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 8 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`

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

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 7: 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 8: This call-site would be considered polymorphic if we only considered the type of the receivers: `StayConstraint` and `EditConstraint`. The call-site can be monomorphised with the knowledge that both receivers resolve to the same type of target `UnaryConstraint`.

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, monomorphising 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 3.2.2).

In Figure 9, 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 3.2.5).

Bootstrap and Application phases. Initialisation phases often differ from the rest of the execution, for instance, because objects and classes need to be loaded and initialised. In our analysis, we distinguish bootstrap and application phases. During the bootstrap phase, the TruffleRuby core libraries are loaded and initialised. The application phase starts when the runtime starts to load and execute the user code. We analyse the impact of the bootstrap phase on call-site

A case of splitting

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

Figure 9: 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.

behaviour in Section 3.2.6. Where not otherwise stated, we include data from both phases.

3.2 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 optimisations, call-site behaviour over time, and the differences in use of methods and closures.

3.2.1 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 of statements (Stmts in Table 1) 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.

We give a general overview of our benchmark set in Table 1. Our benchmark set contains 74 benchmarks. For brevity, we aggregated the results of several series of benchmarks, e.g., the ChunkyCanvas, ChunkyColor, and PsdCompose, PsdUtil benchmark series, since the different benchmarks they contain behave similarly, both in terms of structure and behaviour.

Table 1: Overview of our benchmark set. 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 behaviour, and their values have been averaged. The three colours correspond to the benchmark category (top to bottom: megamorphic, polymorphic, minimally-polymorphic).

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 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.

Research Question 1 *Can these benchmarks be classified according to their call-site behaviour?*

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****.*

We empirically set up these three categories of benchmarks such as they each display homogeneous call-site behaviour. This should ideally improve the readability of our results, since it limits the impact of outliers. We set the 1.5% threshold between polymorphic and minimally-polymorphic benchmarks by observing the difference in structure between the different benchmarks. The minimally-polymorphic benchmarks are often smaller in terms of number of statements (below 16,000), and display an homogeneous behaviour (see Appendix A). Any of the polymorphic benchmarks behaves differently compared to minimally-polymorphic benchmarks, if we consider the number of statements, number of executed call-sites, and the impact of optimisations on the lookup cache contents. This threshold may need to be re-evaluated if other polymorphic benchmarks were to be added in the set in the future.

Table 1 and the remaining tables of this Section show the benchmark set in the order of these three categories. The first category contains the megamorphic benchmarks e.g., BlogRails, that contain at least one megamorphic call-site (Table 2 contains the detailed volume of polymorphic and megamorphic calls). 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 Appendices A.2 and A.3.

From the perspective of call-site behaviour, 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 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 majority of our 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 (Sarimbekov et al. 2013). BlogRails, ERubiRails, Sinatra and HexaPdf are outliers in that regard, since they display a higher amount of polymorphic call-sites. Compared to the other benchmarks of the set, we observe these four benchmarks rely on external Ruby libraries, which is likely having an impact on the amount of polymorphic calls due to code re-use. MailBench, displaying 96.4% of monomorphic calls, also rely on many external libraries. The use of external libraries in real-world workloads is common, therefore we argue these outliers in our benchmark set are likely to better represent real-world applications.

3.2.2 Impact of the existing call-site optimisations on call-site behaviour

As described in Section 2.4.3, TruffleRuby uses splitting and the elimination of call target duplication to optimise call-sites. Since previous studies have not considered how effective these optimisations 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 categories 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 the duplicates (columns four and five). Appendix A.3 provides additional data from a static perspective, showing the impact of these optimisations on the number of polymorphic and megamorphic call-sites.

Table 2: Impact of the elimination of target duplicates on the amount of polymorphic and megamorphic calls (see Columns 2 and 3 for the initial amount of calls). Eliminating target duplicates in the cache is very effective at reducing polymorphism: it is generally reduced by 73.5% (median, see Column 4), except for RedBlack, that has a lower volume of duplicates of less than 8%. Minimally-polymorphic benchmarks excluded for brevity.

	Number of calls		**Change 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%

Our results show that 73.5% of polymorphic calls are subject to target polymorphism. Eliminating target duplication reduces the number of polymorphic calls significantly. The benchmark that benefits most is LiquidParseAll, which parses Liquid HTML templates. It has 87.4% of its calls monomorphised. 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.

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, on the right, indicates the number of method duplicates created. Notably, the table shows that all remaining polymorphic calls are monomorphised by splitting.

The benchmarks with a large number of polymorphic call-sites (see Table 17 in Appendix) usually have a high amount of splitting, as we would expect considering Truffle's splitting heuristic (see Section 2.4.3). 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 monomorphised by splitting (see Appendix A.3). 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 3.2.3, we discuss why these benchmarks may experience splitting more than twice, even though they had only a few remaining polymorphic call-sites.

Table 3: Impact of splitting on the number of polymorphic and megamorphic calls (see Columns 2 and 3 for the volume of megamorphic and polymorphic calls). Columns 4 and 5 shows the proportion of these polymorphic and megamorphic calls being monomorphised by splitting. Observe that nearly all calls have been monomorphised by splitting for nearly all benchmarks. Minimally-polymorphic benchmarks excluded for brevity.

	Number of calls		**Change 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 behaviour. We observe these two optimisations are very successful at monomorphising polymorphic call-sites, as well as eliminating megamorphism.

Table 4 shows how the two optimisations 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, with only few caches remaining 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, but 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 optimisations, reduced to two after eliminating duplicates, and completely monomorphised after splitting. A closer look at the distribution of receivers shows that it remains unchanged at any optimisation stage, with at least 75% of calls being monomorphic only.

Table 4: Impact of the elimination of target duplicates and splitting on the maximum lookup cache size. Both optimisations reduce the maximum cache size, and turn almost all caches monomorphic, even when they held many targets initially. Minimally-polymorphic benchmarks excluded for brevity.

	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

Observation 4 *Splitting in combination with addressing target polymorphism is*

effective at monomorphising 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 monomorphised.

3.2.3 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 next question is:

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

Chambers and Ungar (1989) stated that the aim of splitting is to monomorphise polymorphic call-sites. Indeed, Truffle's heuristic (see Section 2.4.3) 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 splits, and how splitting influenced the lookup cache state of the split call-sites. Quite unexpectedly, if we aggregate our results across the whole benchmark set, we observe that 90.5% (min. 60%, max. 94%)[3] of splitting happens on monomorphic call-sites that remain monomorphic with the same target after splitting, which suggests over-splitting may have occurred. The detailed results by benchmark is available in the last column of Table 5.

[3]Median across all benchmarks

Table 5: Change in the number of lookup cache entries after splitting. The results indicate that between 60% to 94% of splitting occurs in call-sites that had 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%

We inspected several of these cases manually and identified that they result from recursive splitting (see Section 2.4.3).

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 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.*

3.2.4 Methods versus closures

Ruby provides both methods and closures to the developer (see Section 2.4.2). While we have focused so far only on methods, this section analyses the call-site behaviour of closures, asking the question:

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

Table 6 summarises the polymorphic behaviour of closures in our benchmarks. Columns 2 to 4 repeat the data on polymorphism 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 proportion in % 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.

Method calls are much more frequent than closure calls, by a median factor of 5.96 (min. 0.03×, max. 10996×) 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 for closure call-sites.

Table 6: Overview of our benchmark set, comparing method calls and closure calls. Overall, an average of 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%

With the most polymorphic benchmark Sinatra 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.[4] 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 2.4.3), which are also used for methods. This setting was subsequently also adopted as the new default by TruffleRuby's developers. In this context, split-induced monomorphisation 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.

Observation 6 *The polymorphic behaviour of closures and methods is similar, but methods calls are more frequent, and closure calls tend to be slightly more polymorphic. The benchmarks categorised as megamorphic and polymorphic regarding methods calls are not necessarily categorised as such if we consider the polymorphism in closures, for which the behaviour seems homogeneous across the whole benchmark set.*

3.2.5 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 initialisation phase, where initialisation and setup code is executed. This may leave lookup cache entries behind for types that are not used later on. To investigate more broadly, we ask the following question:

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

[4]Unlike methods, that may get split when they contain a lookup cache that turns polymorphic, following Truffle's splitting heuristic.

Table 7: Splitting succeeds at completely monomorphising 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%

To answer this question, we analyse the behaviour 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 monomorphise 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. For the remaining call-sites, we plot the calls to the different targets over time, which we derive from the line number in the log file (see Section 3.1.3). A dot represents a call. It is coloured according to the method implementation that was activated, represented on the y-axis. The vertical black line delimits the switch from bootstrap to application phase (see Section 3.1.3). The behaviours we observe can be categorised into three distinct call-site behaviour patterns that repeat across our benchmarks.

Mirroring pattern. Different targets at a call-site may display similar behaviours throughout execution. Figure 10 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.

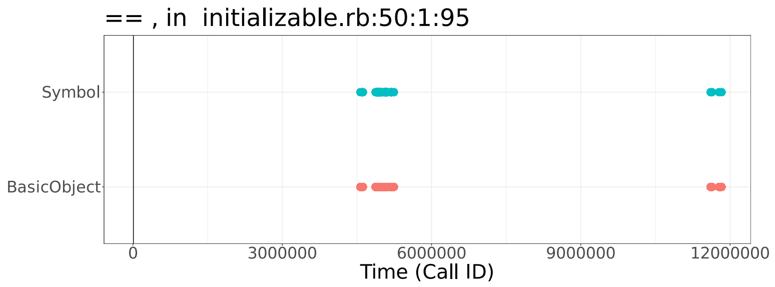


Figure 10: Mirroring pattern in ERubiRails. The plot represents the evolution over time of the calls to the == method from line 50 of initializable.rb. This call-site sees two different implementations of `==`, one for Symbol and one for BasicObject. We see that, at this call-site, the calls to these two targets exhibit the same behaviour, i.e., mirror each other. One dot represents one call.

Phase behaviour. Phase behaviour is characterised by a visible change in the pattern over time. We distinguish here two variants, behaviour at the level of a single target and behaviour 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

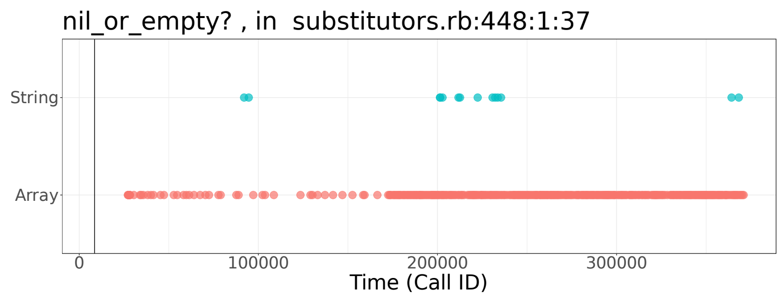


Figure 11: Phase behaviour for the Array target in ADConvert. The plot represents the evolution over time of the calls to the nil_or_empty? method from line 448 of substitutors.rb. The frequency of calls to the Array target increases noticeably about halfway through the benchmark. One dot represents one call.

called at a call-site.

Figure 11 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.

Initialisation pattern. The Initialisation pattern is a specific example of the Phase behaviour 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 behaviour changes, for instance to use a different set of targets. Figure 12 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 behaviour. An example is shown in Figure 13. 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 (Perelman et al. 2006; Nagpurkar 2007),*

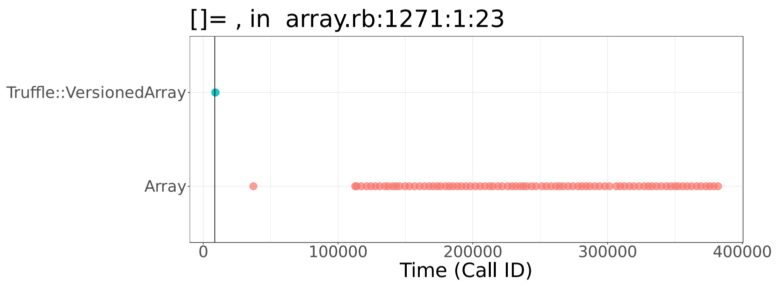


Figure 12: Initialisation pattern in LiquidRenderBibs. The plot represents the evolution over time of the calls to the []= method from line 1271 of array.rb. This is an extreme example with a tiny number of calls to VersionedArray initially, after which only Array is called. One dot represents one call.

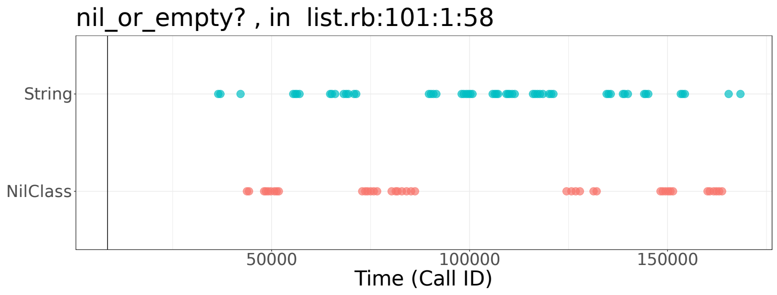


Figure 13: No Apparent pattern. The plot represents the evolution over time of the calls to the nil_or_empty? method from line 101 of list.rb. This call-site in the ADLoadFile benchmark displays no apparent pattern. One dot represents one call.

some call-sites exhibit an initialisation pattern in our benchmarks. In addition, we see patterns of mirrored behaviour, phase behaviour, and call-sites without any apparent pattern. One dot represents one call.

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 14. It displays partial mirroring across the three targets and shows phase behaviour, switching halfway through.

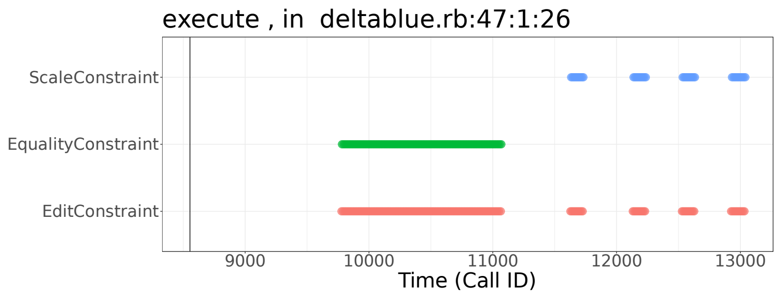


Figure 14: Multiple patterns. The plot represents the evolution over time of the calls to the execute method from line 47 of deltablue.rb. This call-site in the DeltaBlue benchmark displays several call-site behaviour patterns at once. One dot represents one call.

3.2.6 Impact of bootstrap

We can see in the plots of Section 3.2.5 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 used 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 adds to polymorphism.

In our benchmark set, the bootstrap phase has only a minimal impact on the degree of polymorphism. Only four call-sites see an increase of their cache size. However, it affects 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 initialisation can happen on demand. This blurs the boundaries between initialisation and computation phases and makes it harder to consistently identify the switch between the two. Bootstrapping on the other hand is finished once the user code starts executing. However, the small number of call-sites impacted by the bootstrap phase suggests that the phase bounds might need to be re-assessed.

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.*

3.2.7 Limitations

In this section, we discuss some limitations of our study of common assumptions about run-time behaviour.

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

Splitting specialisations TruffleRuby splits method call-sites once they turn polymorphic. However, methods are also split when there are other types of polymorphic operations, e.g., arithmetical 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 behaviour, and are therefore ignored by our instrumentation. However, we confirmed that only a negligible number of splitting operations are caused by these omitted operations.

Handling larger codebases Future refactoring of our processing scripts are necessary to be able to analyse larger codebases. At the moment, the scripts we use to process the traces and produce statistics cannot analyse traces larger than 10GB.

3.3 Insight and recommendations

We analysed 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 optimise language implementations used in production. We found that the majority of our benchmarks display a low degree of polymorphism, with only 1.9%[5] (min. 0.6%, max. 4.4%) call-sites being polymorphic, which is roughly in line with the literature. However, we also see that our industrial benchmarks display more polymorphic behaviour. Benchmarks of this scale have been under-represented 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 monomorphised (see Section 3.2.2). 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 uninitialised lookup caches) and profiling of the split copies, as well as additional JIT compilation costs, and thus, potential prolonged warmup. For large codebases, this can be a performance issue for JIT compilation.

While we might expect splitting most frequently to monomorphise 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

[5]Median, excluding the four outliers ERubiRails, BlogRails, HexaPdf and Sinatra.

an identical state (see Section 3.2.3). By reducing the overall amount of splitting, we limit such cases of oversplitting with Context-Guided Splitting. We further detail this new splitting heuristic in Chapters 4 and 5.

3.3.1 How the splitting strategy may encourage oversplitting: an example

In Chapter 2, we described a splitting use-case in a Ruby program in Figure 5. As is, it already displays the kind of oversplitting uncovered in Section 3.2.3: the two methods `method_two` and `method_three` are split as part of a recursive split. This split occurs as follows: `method_three` contains a polymorphic call-site. The split effectively monomorphises its lookup cache. Since `method_three` is called from a single location (line 7), the split is propagated in the caller chain, to `method_two`. However, `method_two` contains a single call-site (line 7) that stays monomorphic. This is a first kind of oversplitting: monomorphising a call-site that is already monomorphic.

Now, let's add an additional call at line 18 in our small Ruby program, now pictured in Listing 3.1. By default, this call-site points to the original `method_three` implementation, which now contains polymorphic lookup caches due to the previous executions of `method_three` at line 7. To prevent the system from executing a method holding polymorphic caches, this new call to `method_three(ClassA.new)` at line 18 triggers a new split of the `method_three` method.

In that case, we know that `method_three` has previously been split and specialised against `ClassA`, which means that this new split will generate a fresh duplicate version of `method_three`, which will eventually be specialised against `ClassA`, though the first specialised `method_three` is likely to be still part of the program. This is a second kind of oversplitting: when splitting happens on a method that has already been split, and for which the existing duplicate could have been re-used.

Listing 3.1: A new call to `method_three` is added line 18. It triggers an additional splitting of `method_three`, generating a clone of `method_three` specialised against `ClassA`. This can be considered as oversplitting because this clone has already been generated previously at run time.

```
1  def method_one()
2    method_two(ClassA.new)
3    method_two(ClassB.new)
4  end
5
6  def method_two(arg)
7    method_three(arg)
8  end
9
10 def method_three(arg)
11   arg.foo()
12 end
13
14 for i in 1..3 do
15   method_one()
16 end
17
18 method_three(ClassA.new)
```

Let's now see how these two kinds of oversplitting are realised in the program representation, with Figures 15 and Figure 16.

Type 1 of oversplitting: monomorphising monomorphic call-sites. In the Figure 15, the two methods circled in red are monomorphic, i.e., they only contain monomorphic lookup caches. However, they got scheduled for splitting as part of a recursive splitting action, that propagates splitting in the caller chain in an attempt to locate and eliminate a source of polymorphism in the program.

This may seem counterintuitive, as we would expect splitting to only apply to polymorphic methods, so that ideally their caches stay monomorphic until the end of the execution. However, the data responsible for turning a cache polymorphic might flow through methods, as arguments for instance, without impacting the cache status of these methods, but eventually impacting the cache status of methods deeper in the caller chain. Therefore, splitting a monomorphic

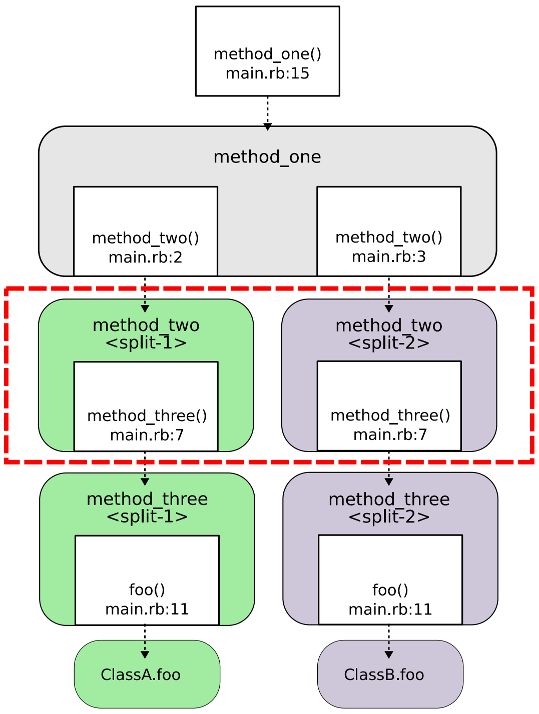
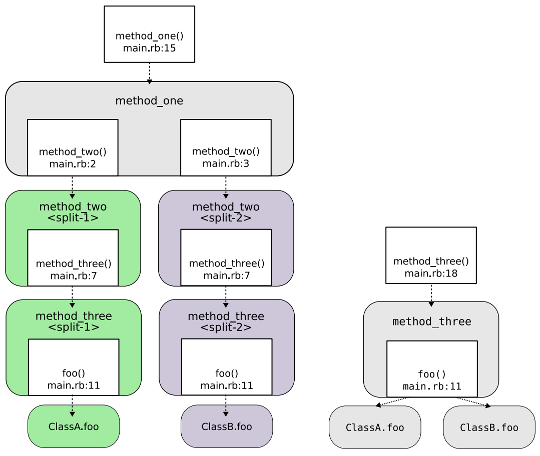


Figure 15: The recursive split decouples the method calls so that the two newly split methods are specialised against `ClassA` or `ClassB`. While necessary to eliminate the source of polymorphism in the program, it leads to the double split of the `method_two` that only contains monomorphic caches (circled in red).

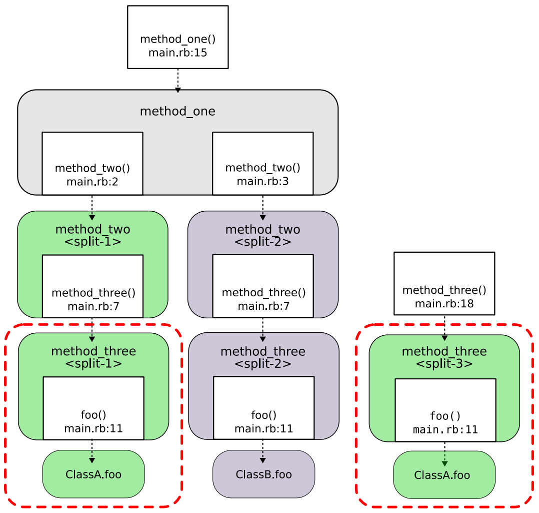
method in this context makes sense to effectively decouple the method calls such as the methods are likely to stay monomorphic.

Once a monomorphic method has been scheduled to be split, then the call-target it is pointing to is flagged as needing to be split, as described in Section 2.2.2. This flag will not be reverted for the rest of the execution. This means that the flagged call-target is going to be split the next time the method is executed, but it also means that any later call to this method, from a different call-site, will trigger a new split of this call-target.

Type 2 of oversplitting: splitting a previously split method. The modified example displayed in Listing 3.1 shows a use-case for another kind of oversplitting. In that case, the method that is split is polymorphic, but splitting generates a fresh method that will end up having the same specialisation as a previously split method.



(a) Before splitting the `method_three` call-site on the right.



(b) After splitting the `method_three` call-site on the right.

Figure 16: Before splitting, the call-site line 18 points to the previous, polymorphic `method_three` target (left). After splitting, the call-site points to a newly generated monomorphic method, that will eventually be specialised the same as `<split-1>` (see circled nodes).

Figure 16 shows the program representation before and after the split of the `method_three` call-site on line 18. On the left hand side of Figure 16a, the program contains the two split and specialised methods generated by the split of the second call to `method_one` (see line 15 in Figure 3.1). The call to `method_three` at line 18 is yet to be executed. Once this call-site is executed and therefore split, the method is duplicated so its cache gets monomorphic. Eventually, once the call is resolved, the cache contains a single entry for `ClassA.foo`, as depicted in Figure 16b. This extra split is triggered to prevent using a polymorphic cache. However, the data flowing in the program is similar, which means the `method_three` call originating from line 2, and the `method_three` call from line 18 both have arguments of the same type `ClassA`. This leads to calling the same implementation of `foo`. Considering no other specialisation was impacted, it means that the red-circled methods in Figure 16b are equivalent, leading to method duplication in the program representation.

From this perspective, the splitting triggered by the call on line 18 can be considered as unnecessary if there were a way to share already split methods.

3.3.2 Impact of (over)splitting on performance

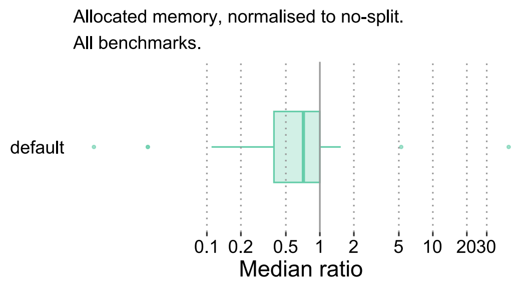
From an engineering perspective, isolating the effect of oversplitting is complex. Splitting monomorphic call-sites is part of the whole splitting process, and eliminating this aspect would equate to redesigning the splitting approach to a large extent, which would in turn make it difficult to pinpoint the causes of a potential performance change. As a first step towards this direction, we start by assessing the impact of the current splitting strategy on different aspects of performance: execution time, heap-allocated memory and time spent in garbage collection, and finally time spent in compilation, and compare those metrics to a case in which all splitting has been disabled. The detailed benchmarking setup, as well as an an explanation of how the results are computed, is available in Section 5.3. We use the same benchmark set as in Chapter 5 and we show that splitting has a positive impact on memory-related metrics (up to -100% of heap memory used[6]) and overall execution time (decreasing the time by up to 91.9%), while increasing the median time spent in compilation by up to 3.32× (min. -85.4%, max. 3.32×). We then extrapolate the possible performance benefits of addressing oversplitting from these results.

Splitting positively impacts memory usage We first assess the impact of splitting on two metrics related to memory usage: the amount of memory (in megabytes) allocated in heap memory for the current thread, since our benchmarks

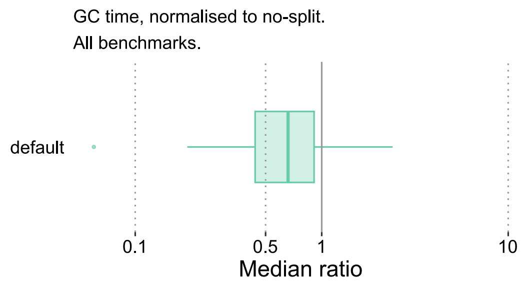
[6]Splitting, by monomorphising a program, enables further optimisation of a program. See in "Splitting positively impacts memory usage" for a concrete example.

are single-threaded,[7] and the time spent in garbage collection.[8] The first metric covers the garbage collected heap of Java objects. Since we use a version of Graal that has not been compiled ahead of time (see Section 5.1), the heap contains both the GraalVM compiler and application objects. We did not assess the impact of splitting on micro-architectural issues such as, i.e., L1 instruction-cache misses.

We assume that splitting, by duplicating methods, may have a negative impact on heap memory usage, though this effect might be compensated by the extra optimisations being enabled by splitting itself, such as scalar replacement and boxing elimination based on type-specialised code paths.



(a) Splitting overall reduces the volume of allocated memory



(b) Splitting helps decrease the time spent in garbage collection

Figure 17: Enabling the DEFAULT splitting heuristics in TruffleRuby lowers the amount of memory allocated in the heap compared to a case when splitting is disabled (left). This has direct consequences on time spent in garbage collection (right), where splitting performs better too. Logarithmic scale (Lower is better).

The two figures part of Figure 17 display the benefits of splitting on heap-allocated memory (on the left, Figure 17a) and on the time spent in garbage collection (on the right, Figure 17b). They show that splitting improves memory usage by decreasing the volume of heap-allocated memory and the time spent in garbage collection. Across our benchmark set, the median volume of allocated memory in the heap is reduced by 29.9% (min. -100%, max. 46.91×), reaching a

[7]`https://docs.oracle.com/en/java/javase/11/docs/api/jdk.management/com/sun/management/ThreadMXBean.html#getCurrentThreadAllocatedBytes()`

[8]`https://docs.oracle.com/javase/8/docs/api/java/lang/management/GarbageCollectorMXBean.html#getCollectionTime--`

maximum decrease of 100% for NBody, compared to when splitting is disabled. We observe such a sharp decrease in allocated memory in some of our micro-benchmarks. Splitting enables more monomorphisation of programs, which in turn enables further optimisations. For this outlier specifically, the monomorphisation caused by splitting helps the escape analysis[9]. The NBody benchmark is dominated by calculations on Double values, which are normally boxed. However, because of the monomorphisation enabled by splitting, the escape analysis can remove all allocations of Doubles, so that the benchmark does not use any heap memory anymore, which explains the change of heap size is -100%. Likewise, the median time spent in garbage collection is reduced by 50% (min. -100%, max. 4,000,000$\times$), with a maximum decrease of 100%, [10] for Bounce as well.

The assumption is that splitting, by monomorphising the program, helps trigger inlining during JIT compilation, which in turn will allow other optimisations to be applied. Notably, partial escape analysis is triggered and helps eliminate redundant allocations through scalar replacement. This decrease in allocations likely leads to a decrease in heap-allocated memory.

Splitting positively impacts execution time. In addition to improving memory usage, splitting has a positive impact on execution time. Figure 18 compares how the execution time in our benchmark set differs when splitting is enabled or disabled and shows that, overall, the median execution time is reduced by 13.9% (min. -91.9%, max. 70.1%) once the standard splitting heuristics are enabled in TruffleRuby. The execution time is decreased by up to 91.9% for ImageDemoSobel. While infrequent, some of our benchmarks see their execution time greatly increased

[9]See `https://chrisseaton.com/truffleruby/seeing-escape-analysis/`. Last accessed the 18/02/2025.

[10]We observe extreme outliers when monitoring the time spent in garbage collection. A decrease of 100% is observed when comparing a non-zero baseline median (here NO-SPLIT) to a 0ms median (with DEFAULT, no time recorded in GC). Sometimes, we observe very important increase in GC time. This is caused when we did not record any time spent in GC for a benchmark for our baseline, but observed a median greater than 0 for the splitting strategy we compared it against.

when splitting is enabled; MatrixMultiply and Queens for instance, experience respectively an increase of 70.1% and 10.6% of their median execution time.

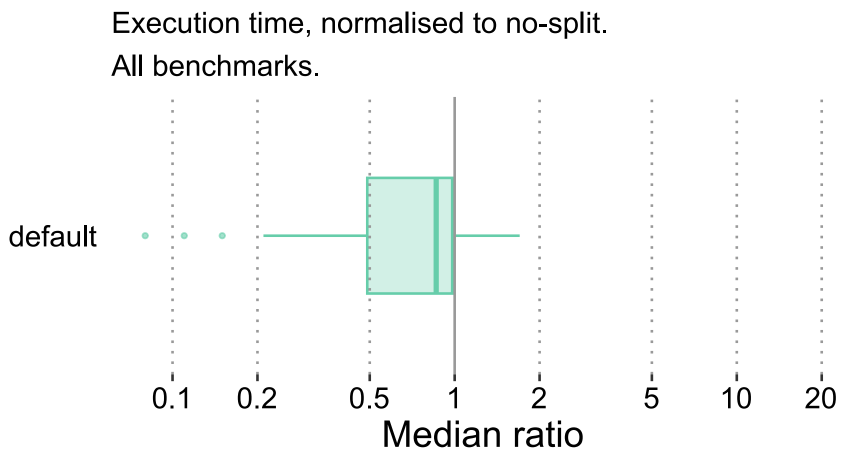


Figure 18: The median execution time is reduced by 13.9% when splitting is applied, compared to a case where splitting has been disabled.

Splitting does not impact compilation time. Splitting leads to the generation of method clones, that may require to be compiled. We therefore check the impact of enabling splitting on the time spent in compilation at run time. Enabling splitting has a marginal effect on compilation time, as shown in Figure 19. Indeed, we observe a 0.42% increase of the median compilation time (min. -85.4%, max. 3.32×). This small increase is consistent across the whole benchmark set, except for a handful of benchmarks. Mandelbrot, ImageDemoSobel, and BlogRails experience a decrease in the time spent in compilation at run time, of more than 74.7%. On the other hand, the benchmark PsychLoad is the one experiencing the largest increase in compilation time, with a 3.32× slow down.

3.3.3 The benefits of eliminating oversplitting: lower heap memory usage, lower compilation time

The existing splitting strategy used in TruffleRuby has a positive impact on run-time performance, e.g. execution time and overall heap memory consumption. However,

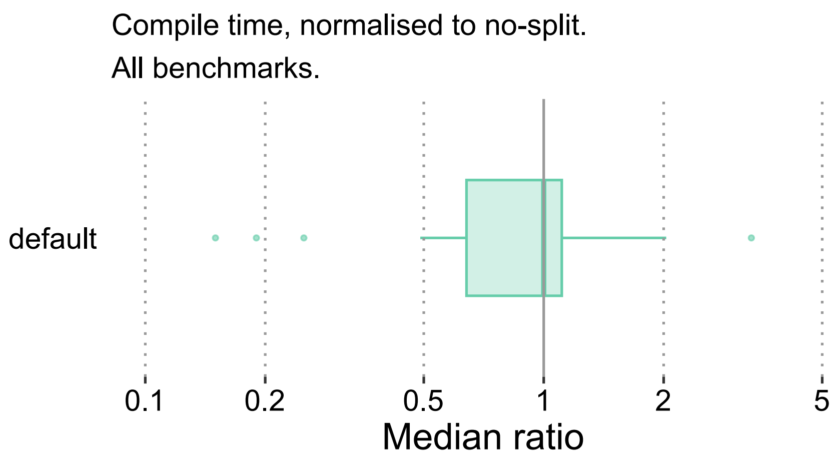
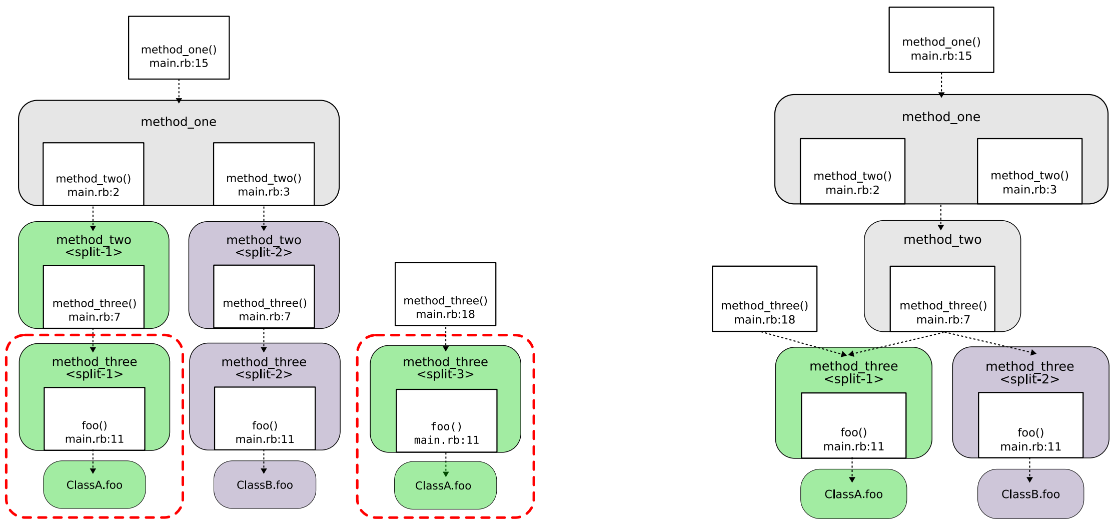


Figure 19: The median compilation time is marginally increased by 0.42% when splitting is enabled, compared to a case where splitting has been disabled.

we showed in Chapter 3 that splitting is generally applied on monomorphic call-sites, and that it also generates method clones that have already been generated. So, we argue that better guidance of splitting decisions in TruffleRuby may improve the performance of splitting. More specifically, a new heuristic that triggers splitting less often could lead to:

- Reduced heap memory consumption, and therefore less time spent in garbage collection, because fewer method clones are produced;
- Reduced compilation time, which may lead to improved start-up performance, as there are fewer compilation units to process;
- Improved–or unchanged–execution time, as less run time is spent in splitting-related actions.

Figure 20 presents a side-by-side comparison of a program with and without oversplitting. Without preventing oversplitting, in Figure 20a, the call-site line 18 in the `main.rb` file, noted `main.rb:18`, and depicted as a white rectangle in the left figure points to a clone of the `method_three` target (circled red on the right of Figure 20a), while an equivalent method is already part of the AST



(a) AST with existing splitting heuristics

(b) Ideal AST structure if over-splitting is addressed

Figure 20: A same program without preventing oversplitting (left) and after having eliminated oversplitting (right). Addressing oversplitting would eliminate duplicated nodes, which means the call-site line 18 would then refer to the already generated and specialised method. The same reasoning applies on the `method_two` call-sites.

(circled red on the left). Addressing oversplitting would eliminate duplicated nodes, which means the call-site line 18 would then refer to the already generated and specialised method. The same reasoning applies on the `method_two` call-sites. Ideally, if we were to address oversplitting, the resulting AST would be structured as pictured in Figure 20b. The two call-targets for `method_two` would be replaced by a single call-target, as only the data that flows through them is different: their arguments are of a different type, but this does not impact the implementation of the method. Typically, the lookup cache they hold for the `method_three` call-site is monomorphic, with the same entry. In addition, the two `method_three()` call-sites would then point to the same call-target in this ideal AST, considering they currently point to two different clones that have however been specialised in the same way.

In this example, the AST has been simplified to a great extent for readability, only keeping call-nodes and call-targets, and we therefore only observe a reduction of four nodes. In reality, the AST is much richer and avoiding the duplication of methods would avoid the duplication of many more AST nodes.

Considering that our experiments about behaviour in this Chapter suggest that on average 90.5% (min. 60%, max. 94%) of splitting occurs on monomorphic methods, and that they lead to duplicated methods in the program's representation, we therefore expect that reducing oversplitting may positively impact performance in TruffleRuby. In the next Chapters, we describe how Context-Guided Splitting, a novel splitting strategy, may address the second type of oversplitting that leads to duplicated method in the program representation. The principles of Context-Guided Splitting are discussed in Chapter 4. We implemented this splitting strategy in TruffleRuby, and the following Chapter 5 describes the impact of this approach on performance.

Chapter 4

Guiding splitting decisions using contextual insight: general principles

Our analysis shows that, albeit very effective at monomorphising a program, splitting may be triggered too often if only guided by receiver types. We call this phenomenon oversplitting and identified two oversplitting use-cases: either when a monomorphic call-site gets split, which means we unnecessarily monomorphise an already monomorphic call-site, or when a method duplicate is generated through splitting but ends up being identical to a previously generated method, which means the runtime could have re-used the previous method and avoided the split.

Such oversplitting may have an adverse effect on performance and may increase compilation time and memory usage. However, addressing both use-cases of oversplitting simultaneously is not trivial. Notably, splitting monomorphic call-sites is an integral aspect of the current splitting strategy to allow the elimination of the source of polymorphism (see Section 2.4.3). We therefore focus on avoiding splitting when a suitable method clone has already been generated. We inspire from the contextual dispatch approach proposed by Flückiger et al. (2020) and present the principles and implementation of a alternative splitting strategy in TruffleRuby guided by contextual insight. The main goal of this approach is to re-use previously generated clones of methods by dispatching to them based on contextual information. The expected benefits are a reduction of the number of splits, which we expect to lead to lower compilation times and better memory usage, thanks to method reuse. While not designed specifically to address monomorphic splitting, we expect a side-effect of this approach will be to also reduce–but not to eliminate–the number of monomorphic methods being split.

We name this alternative splitting heuristic, *Context-Guided Splitting*. In this Chapter, we describe the main elements of our approach, namely contexts and dispatch points (Section 4.1), and how we implemented it in TruffleRuby. We explain how we use arguments' types as building blocks for a context (Section 4.2.1), how we set up dispatch locations (Section 4.2.2) and how we revert to default splitting when a context misprediction occurs (Section 4.2.3). We detail the impact of our Context-Guided Splitting approach on performance and behaviour in Chapter 5, with and without handling context mispredictions. We show that it mainly benefits megamorphic benchmarks, for which it reduces the time spent warming up, reduces the size of the AST and the time spent in garbage collection. The standard splitting heuristic in TruffleRuby tends to benefit to a larger set of benchmarks, including the ones displaying a low level of polymorphism, where it speeds up peak execution time and decreases the volume of allocated memory.

4.1 Sharing specialised methods: the general principle

In Section 3.3.3 in the previous Chapter, we described an ideal program representation for our splitting use-case where the extraneous nodes generated by oversplitting had been eliminated (see Figure 20b). We argue that modifying the current splitting heuristics used in TruffleRuby may successfully help reaching this ideal state, or a close-enough state.

We follow the approach of Flückiger et al. (2020) on contextual dispatch and propose Context-Guided Splitting as a way to address the oversplitting issues identified in TruffleRuby. With this approach, splitting decisions are guided using further information available at the moment of the call, i.e. contextual information, in addition to the information used by default, i.e., if the lookup cache turns polymorphic. In their work, Flückiger et al. (2020) use assumptions about the arguments, e.g. whether they have been evaluated, whether they are objects, and assumptions about the invocation itself, to represent a context. We describe our own representation of a context in TruffleRuby in Section 4.2.1.

The heuristics guiding a splitting decision in TruffleRuby are simple: as soon as the lookup cache of a call-site turns polymorphic, the call-site is flagged for splitting, and will be split right before its next execution.[1] It means that the initial trigger for a splitting action is this change of state of the lookup cache. We use contextual insight to decide, once a method has been flagged for splitting, whether to generate a new uninitialised clone through splitting, or to dispatch to a previously generated clone that shares the same context as the current call.

In practice, Context-Guided Splitting adds a level of indirection, i.e. a dispatch point, that uses contextual data to cache the clones generated by splitting so they can be re-used at a later time if the same context is encountered. This is represented in Figure 21. The new level of indirection is represented by the rectangle stating "check contextual info": this is when we check for a context match. Following the same example depicted in Figure 20, this would translate as follows, as shown in Figure 22. The `method_two` and `method_three` are clones

[1]Monomorphic call-sites may end up being split as part of recursive splitting, as explained in Section 3.3.1. Some other cases disqualify a method from being split, for instance, if the total number of AST nodes created via splitting exceeds a given threshold.

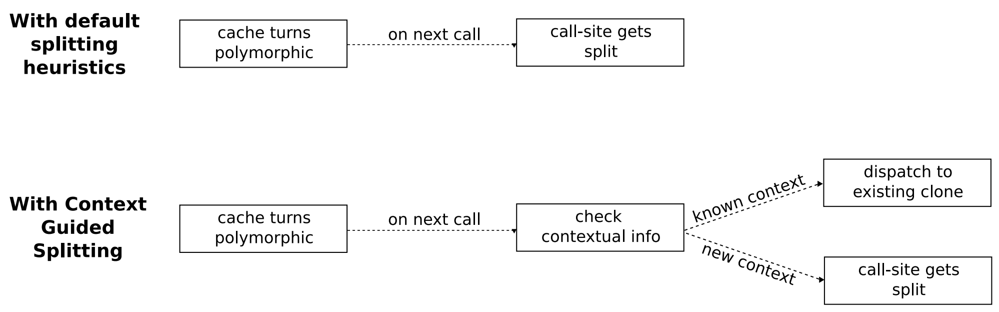


Figure 21: Our Context-Guided Splitting approach relies on the lookup cache state. The final splitting decision may however be aborted in favour of a dispatch to a previously generated clone, if the current context of the call has previously been recorded.

produced from a previous split, and are now shared between several call-sites. Two dispatch points, represented by coloured squares in the Figure 22, have been added dynamically; they hold previous contextual information (coloured in green and purple in the Figure), and dispatch to a previously generated method matching this contextual information. Adding this dispatch point should have a marginal impact on performance: it is only used when a splitting occurs, and not every time the call has to be executed. When a splitting decision has to be made, there are three possible outcomes: splitting does not occur at all, and the execution continues as is; split is decided, and the method is cloned; and split is decided, but it is possible to dispatch to a previous method clone, and so dispatch happens rather than splitting. In the latter case, there is an extra indirection induced by the dispatch, but it only occurs once to find the correct implementation to point to. Further calls to this method will not go through this indirection, except if a new splitting decision has to be made.

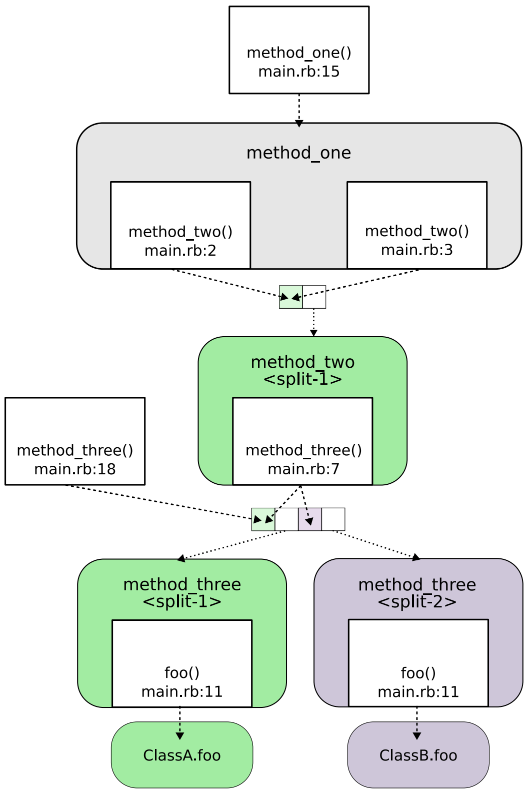


Figure 22: Program representation after Context-Guided Splitting was applied. The two `method_two` call-sites share the same context, and are therefore pointing to the same `method_two` clone. The calls to `method_three` sees two different contexts (depicted as a green, and purple squares), and dispatch to two different clones of `method_three`.

4.2 Implementation of Context-Guided Splitting in TruffleRuby

We implemented Context-Guided Splitting in TruffleRuby version 22.3.1, using GraalVM Community Edition. In this Section, we detail the main implementation aspects of our approach, starting with a description of what is a context (4.2.1). We then explain the dispatch mechanism (4.2.2) and how we deal with potential context mispredictions (4.2.3).

4.2.1 Definition of contexts relevant for Ruby

A Context C is a mapping from (X, Y) to Z. X is a list of argument types, Y is a boolean representing the validity of the context, and Z is the call-target, i.e., the method implementation associated with this context. A context is valid if Z does not contain polymorphic lookup caches (see also “Context misprediction”, Section 4.2.3). A context aims at representing a part of the program’s state at the time of the call. The assumption is that this particular context is likely to influence the program structure, run-time specialisations, and call-site optimisations that will be triggered.

There are multiple run-time specialisations in TruffleRuby, many of which rely on type information. Typically, the self-specialisation of nodes of the AST (Würthinger et al. 2012) may rely on the argument types of a call; call-sites use lookup caches, whose number of entries will depend on the type of the receiver of the call. Argument types and return types are profiled and used, e.g. to speed-up execution.

Since argument types are likely to influence many run-time specialisations, we design contexts as signatures that contain the types of the arguments of a method invocation. In addition, we treat blocks specially in our contexts, as they are widely used in Ruby programs and prone to bias our signature. Indeed, blocks are frequently passed as parameters, which means that two different method calls will have the same signature if we only take into account the type of the argument, e.g. a `Block`. To address this issue, we consider the identity of the target method of the block. This identity, in the form of a hashcode, will change if the target method is split. The context we designed can be considered as a run-time typing context. Thus, it is not a classic calling context, which would contain the set of callers leading to the current call.

The context is stored in the form of an object containing a `Long` signature and

Figure 23: Computation of the context signature

```
function hashContext(ts: list of types)
    $ps \leftarrow list\ of\ prime\ numbers$
    return Long: hashType($ts, ps$)

function hashType(ts: list of types, ps: list of prime numbers)
    $signature \leftarrow 0$
    for $i \leftarrow 0$ to $size(ts)$ do
        $j \leftarrow i \mod size(ps)$
        $signature \leftarrow (hash(ts[i]) * ps[j]) + signature$
    return Long: signature
```

a `boolean` holding whether the context is still valid or not (see Misprediction, in Section 4.2.3). It is calculated in the hashContext function written in pseudo-code in Figure 23. It takes a list of argument types, hashes each of these types in hashType, and return the hashed context signature as a `Long`.

The hash function is implemented as hashType, as featured in Figure 23. It hashes an argument type using the `hash` function provided by HotspotVM, that generates 32-bit hashes. This hash is then multiplied by a prime number, depending on the argument position in the list, to limit the amount of collisions. All calculated hashes are added together to form the context signature.

For instance, if a call to `foo(1, 45, "baz", TRUE)` is executed, then the context related to this call will be equal to:

$$hash(int) * 2 + hash(int) * 3 + hash(String) * 5 + hash(Boolean) * 7$$

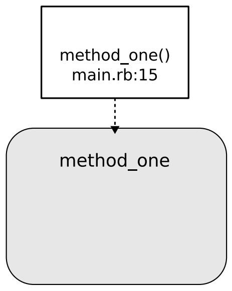
The value of the argument is disregarded for the context computation, but the position of the argument is critical. A call to `foo(573, 832, "bar", FALSE)` would lead to the same signature i.e., the same context, but a call to `foo("doggo",` `3, TRUE, 68262)` will be associated to a different context, whose value is equal to

$$hash(String) * 2 + hash(int) * 3 + hash(Boolean) * 5 + hash(int) * 7$$

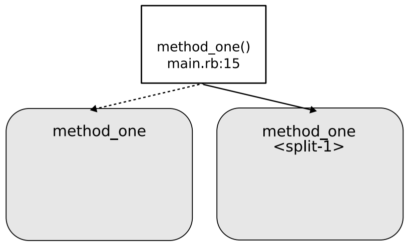
This computation is performed alongside the evaluation of the arguments, when a call is processed to be later resolved. Each call target contains a context signature that is updated to reflect the contextual information of the current call.

We test the robustness of this approach by picking three benchmarks from our set, and verify whether we observe any signature collision. A signature collision occurs when a different set of arguments leads to the same context signature, i.e., the hashed signature described above. We first pick BlogsRails, our largest benchmark, and track 50,413,717 signatures for 1 iteration. Only 0.18% collided, i.e., 0.18% of signatures were associated with more than one set of arguments. We then checked a medium-sized benchmark, AsciidoctorLoadFile, ran it for 150 iterations, and observe that, of 77,787,252 generated signatures, 0.1% collided. Lastly, we checked the benchmark displaying the highest number of context mispredictions (see Section 4.2.3), MatrixMultiply, which did not display any collisions. The collisions we observed in these two benchmarks are all consistently explained by the splitting of blocks. If a block is passed as a parameter to a call, and the block is split, the hashcode of the target method of the block changes. However, the method signature stays the same. The low number of collisions suggests that our hashing approach is effective enough, while being computed at an acceptable complexity.

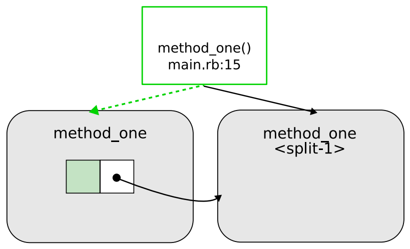
Ideally, the same context would always trigger the same specialisations on a given method duplicate and two different contexts should lead to different specialisations. In practice, some side-effects are not captured by the context and may influence the method state. For instance, user input, the value of fields or globals may all have an impact on the specialisation of a method, or the content of a lookup cache. This means that a single context may need to point to different



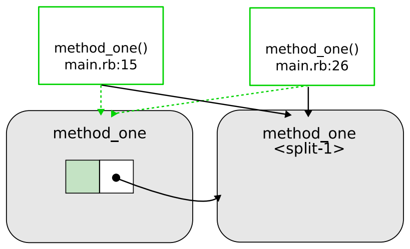
(a) No splitting



(b) After splitting



(c) After Context-Guided Splitting, one call-site



(d) After Context-Guided Splitting, two call-sites

Figure 24: A call-site (represented as a square-cornered rectangle) always keeps a reference to its initial call-target, even after splitting. Context-Guided Splitting uses the initial call-target as a dispatch point to store previously generated clones.

duplicates. We call this case *Context misprediction* and discuss in Section 4.2.3 how we handle it.

4.2.2 Setting up contextual dispatch locations

We use these context signatures at specific dispatch locations, that we call contextual dispatch locations and that are local to a call-target.

A call-site points to a call-target that is executed when the call-site is activated. For instance, say we have a program with one call-site, `method_one()` at the line 15 of a `main.rb` file. We represent it as a white rectangle in Figure 24a. It originally points to the call-target, `method_one`, represented by a grey rounded-edge rectangle.

When this call-site is split, the AST nodes constituting of the call-target are cloned to their uninitialised, i.e. non-specialised state, and the call-site now points to this split call-target, which means this split call-target is the one that will be executed when the call-site is activated from now on. This process is depicted in Figure 24b, where the clone is the rounded-edge rectangle on the right. However, the original, non-split, call-target i.e., the rounded-edge rectangle on the left, is still referenced by the call-site, and will be throughout the whole execution.

We take advantage of this design, and add a contextual dispatch location in this initial call-target. It is shown in Figure 24c, in the form of two small squares in the call-target. The dispatch location consists of a data structure associating contexts (the green square) to previously generated clones for this particular call-target we are setting the dispatch location into. If a call-target is never split, then this data structure is empty. However, if the call-target is flagged to be split, then this data structure is populated.

If a call-target is split for the first time, the resulting cloned call-target is automatically added to the data structure. The context signature currently stored in the call-target and that holds the contextual information relevant to this particular call, is used as a key for this split call-target. In Figure 24c, it means that the green context associates with the clone of `method_one` flagged as `<split-1>`.

Later at run time, if another call-site pointing at this call-target is flagged for splitting, the split will not necessarily happen. Such a case is depicted in Figure 24d, with the addition of a new call-site for `method_one`, at the line 26 of a `main.rb` file, denoted `main.rb:26`. We first check whether the current context is stored in the data structure. If it is, then the call-site will now point to the associated call-target (a previously generated clone), rather than generating a new uninitialised clone of this target. In our example, the two call-sites for `method_one` share the same context, as denoted by their green edges, which means they can both dispatch to the only context that is stored at the dispatch location.

If the current context is not found, then the call-target is split normally, and the association context signature-split call target is stored in the data structure.

4.2.3 Handling mispredictions

We use contextual information to avoid generating new method clones, and instead re-use previously generated clones. We compare the current context of the call with the stored context signatures for a given call-target, and if there is a match, we dispatch to the associated method clone.

This approach is speculative: ideally, we would like to distinguish contexts accurately enough so that two different contexts are always pointing at different method specialisations. However, the specialisation may have been caused by an event that is not captured in the contextual signature. We limit the amount of contextual information that is part of the signatures to ease future comparisons, and some information may not be easily accessible when a context signature is computed. This may lead two call-sites having the same context, i.e. dispatching to the same method clone if Context-Guided Splitting is enabled, but leading to different specialisations of the method. This situation is problematic, because it means the shared method is likely to become "polluted", with, for instance, polymorphic caches instead of monomorphic caches, and therefore less prone to being further optimised.

To detect mispredictions, we first flag as "shared" each call-site contained in a shared method clone, as soon as it is added to the shared cache. We consider a misprediction to occur if any of these shared call-sites is flagged for a future split: it means that one or several of their lookup caches turned polymorphic, suggesting that the current context led to specialisations different from the context we dispatched to. In that case, we invalidate the context we dispatched to. It means that all the shared call-sites associated with this context are flagged as being invalid, and that they therefore should not be dispatched to in the future.

This invalidation comes into effect the next time a context matches with the invalid context. We insert a guard before actually dispatching, which, if triggered, leads to the deletion of the pair `(Context, Method clone)` from the data structure. In that case, a normal split occurs, and the resulting clone is not further stored in the data structure, as we know this context will likely lead to a future misprediction.

Figure 25 shows a snippet of code representing a Ruby program experiencing a context misprediction.

(a) Implementation of foo in ClassA

```ruby
1  def foo(arg)
2    puts arg+" A"
3    return Time.now()
4  end
```

(b) Implementation of foo in ClassB

```ruby
1  def foo(arg)
2    puts arg+" B"
3    return Time.now()
4  end
```

(c) Main class

```ruby
1   def method_one(startIndex, endIndex)
2     a = [ClassA.new, ClassB.new]
3     for i in startIndex..endIndex do
4       e = a[i]
5       e.foo("Hello")
6     end
7   end
8
9   method_one(0, 0)
10  method_one(0, 1)
11  method_one(0, 0)
12
13  [1 , 1].each do |i|
14    method_one(1, i)
15  end
```

Figure 25: This small Ruby program triggers a context misprediction on the fifth call to `method_one`. A method clone of `method_one` is being shared from line 11, but the cache of the `foo` call-site turns polymorphic on the first iteration of the for loop, which indicates a context misprediction.

The first call to `method_one`, on line 9, triggers a single call to the implementation of `foo` from `ClassA`. At this stage, the lookup cache of the `foo` call-site is monomorphic with this entry. The second call-site to `method_one` at line 10 triggers two calls to `foo`: the first one to the `ClassA` implementation, and the second one to the `ClassB` implementation, leading to the lookup cache of `foo` turning polymorphic with two entries, as the entry for the `ClassB.foo` target is added. Following the splitting heuristics in TruffleRuby, it means that `method_one` is now flagged for splitting, since it contains a call-site that turned polymorphic.

The call to `method_one` on line 11 triggers this split. During the split, since we use the Context-Guided Splitting heuristics, we associate the `method_one` clone generated by splitting with the current context signature of `method_one`[2]. This pair is therefore available to future calls to `method_one`.

The first call to `method_one` at line 14 dispatches to this previously generated clone, since the context signature of `method_one` is the same as the call in line 11 (the arguments are of the same type). This dispatch does not trigger any issue, since the lookup cache of `foo` contains the correct entry, i.e., the `foo` implementation of `ClassA`. The second call to `method_one` at line 14 triggers a context misprediction. It calls the `ClassB` implementation of `foo`. The lookup cache of the `foo` call-site contained in the shared version of `method_one` turns polymorphic, which indicates to the runtime that the shared method clone should not be used any more, since it contains polymorphic lookup caches.

This case of misprediction may seem counter-intuitive since the context signatures of the calls to `foo` are different, with the receivers being `ClassA` or `ClassB`. This is explained by the fact that splitting is performed one level earlier in the caller chain, e.g., if a call-site turns polymorphic, its caller is split. This, in turn, means that we contextually dispatch using the context signature of the caller, i.e.,

[2]The context signature is computed as follows: hashcode of Integer * 2 + hashcode of Integer * 3 + hashcode of the RubyBasicObject representing the entry point of the program * 5, where the plain numbers are prime numbers.

the target being split. In our use-case, the caller is `method_one`, which is always called with arguments of the same types, and will therefore be associated with the same context signature.

In our experiments, we saw mispredictions to be common, and will discuss their frequency and their impact on performance in Chapter 5.

Chapter 5

Results

In this Chapter, I assess how Context-Guided Splitting performs against the default splitting approach available in TruffleRuby. I pick up two variations of Context-Guided Splitting: one that ignores context mispredictions, and one that handles this issue.

I explain our benchmarking methodology in Section 5.1, and then focus on the impact of our different splitting strategies on call-site behaviour, i.e., how splitting impacts the degree of polymorphism in the benchmark set and whether Context-Guided Splitting still lead to oversplitting (Section 5.2). I show that Context-Guided Splitting is effective at monomorphising programs, yet still does not completely avoid the splitting of monomorphic methods. From Section 5.3 to Section 5.5, I compare the performance of four different splitting approaches regarding three performance metrics: execution time (Section 5.3), memory usage (Section 5.4) and the time spent in compilation at run time (Section 5.5).

We first observe that megamorphic benchmarks are the ones benefiting the most from splitting, all performance metrics considered. Regarding peak performance, DEFAULT overall speeds up execution time, by 13.9% (min. -91.9%, max. 70.1%), when Context-Guided Splitting leads to a significant speedup of 38.6% (min. -59.2%, max. -7.2%) for megamorphic benchmarks only, though not outperforming default. When focusing on the impact of splitting on startup performance, we notice default reduces the time spent per startup iteration the most, by 13.6% (min. -85.4%, max. 8.09×), and notably reduces the time spent in the first iteration, often critical in a pay-per-use setting. However, context helps the program stabilising faster for megamorphic benchmarks, i.e., exiting the startup phase 2% earlier (min. -21.4%, max. 2.64×). context also speed-ups most the first iteration of the industrial benchmarks of our set based on Shopify's Liquid renderer. Both Context-Guided Splitting approaches reduce the size of the AST compared to default, with context generating 39.6% (min. -43.7%, max. -14.5%) fewer nodes for polymorphic benchmarks and 25.3% (min. -52.4%, max. 74.3%) fewer nodes for megamorphic benchmarks. This does not seem to correlate with the overall allocated memory, where default and mispredicts perform best overall (respectively, 29.9% (min. -100%, max. 46.91×) and 21.7% (min. -100%, max. 8.35×) less allocated memory). The time spent in garbage collection is also reduced: default is the most beneficial for minimally-polymorphic benchmarks, with 42.1% (min. -100%, max. 4,000,000×) less time spent in GC, while mispredicts brings the most important speedups for polymorphic, with a 51.8% (min. -100%, max. 1.9%) decrease, and megamorphic benchmarks, with a 56.6% (min. -96.7%, max. 2.4%) decrease. Our last metric, i.e., the time spent in run-time compilation, is not significantly reduced for minimally-polymorphic benchmarks. However, polymorphic and megamorphic benchmarks see their run-time compilation time reduced the most by context, by 7.5% (min. -41.3%, max. 9%) for polymorphic benchmarks, and mispredicts by 37.5% (min. -78.9%, max. 2.39×) for megamorphic benchmarks).

In the last Section 5.6, we examine context mispredictions and investigate how they may influence performance.

5.1 Methodology

5.1.1 Benchmark set

We use the same benchmark set as in Chapter 3, comprised of 37 Ruby benchmarks. Table 1 in Section 3.2.1 provides the list of these benchmarks as well as some general metrics to characterise them, such as the number of lines of code and number of calls, coverage and insight on their call-site behaviour. A brief description of each benchmark is available in Appendix A.1. We re-use the benchmark categorisation introduced in Section 3.2.1, and present our results following the three categories: minimally-polymorphic, polymorphic and megamorphic benchmarks. We argue that the polymorphic and megamorphic categories are more likely to benefit from splitting, and from Context-Guided Splitting in general, because they display a higher degree of polymorphism (more than 1.5% of their call-sites are polymorphic) compared with the minimally-polymorphic category.

5.1.2 Benchmarking setup

Unless stated otherwise, the benchmarks were run on three different machines using Ubuntu 22.04.4 LTS (kernel 5.15.0-92), and either two 6-core Intel(R) Xeon(R) E5-2620 CPU at 2.40GHz, with 15.4GB RAM or two 6-core Intel(R) Xeon(R) Gold 6209U CPU at 2.10GHz, with 48GB RAM. We use the TruffleRuby 22.3.1 release on top of a custom build of OpenJDK 20.0.2 with JVMCI and GraalVM CE version 22.3.1.1. We disable background compilation, to only allow compilation work to be performed in the main thread and to have more deterministic runs. We also make sure to use a version of Graal that has not been compiled ahead of time in order to more easily monitor memory usage and compilation time.

Our benchmark set contains diverse benchmarks, from simple micro-benchmarks like Bounce to much larger, complex benchmarks like BlogsRails. We adapt the number of iterations per benchmark individually in an attempt to reach a steady state of performance. We use a similar approach to Barrett et al. (2017) to identify when a benchmark exits its warmup phase (see Section 5.3.2 for more details). We use ReBench (Marr 2023) to run the benchmarks and collect execution time per iteration. It is used in conjunction with rebench-denoise, that disables various system settings to reduce system interference that may add noise to our benchmark results. The benchmark runs configuration, including the exact number of iterations per benchmark and the different execution flags used, is available in Appendix B.1 as a ReBench configuration file.

5.1.3 Four different splitting strategies

We compare the performance and behaviour of four different splitting strategies implemented in TruffleRuby:

- NO-SPLIT: a version of TruffleRuby where all splitting has been disabled, i.e. the standard splitting strategy of GraalVM (see Section 2.4.3), as well as the ad-hoc, splitting strategies specific to TruffleRuby applied to a subset of standard library methods (see 2.4.3); this is the baseline configuration we compare against.

- DEFAULT: a version of TruffleRuby where the standard splitting strategy of GraalVM, as well as the ad-hoc splitting strategies of TruffleRuby are preserved. This is the default setting in TruffleRuby.

- CONTEXT: a version of TruffleRuby where the standard splitting strategy has been replaced by Context-Guided Splitting. In this configuration, Context-Guided Splitting may also apply to the targets of the ad-hoc splitting strategies of TruffleRuby.

- MISPREDICTS: the CONTEXT configuration, with an additional strategy to handle context mispredictions (see Section 4.2.3).

5.1.4 Visualisation and description of the results

In this Chapter, we compare the performance of the four splitting strategies and, unless stated otherwise, normalise our results to NO-SPLIT, i.e. a version of TruffleRuby with all splitting disabled.

Our benchmark set is large (37 benchmarks) and diverse, with micro-benchmarks of a hundred line of codes to larger benchmarks of thousands lines of code and more complex behaviour. We provide a visual summary of the results using boxplots, and use either multiplicative ratios and change rates against NO-SPLIT to describe the differences in performance. We use this approach to improve readability and to ease the comparison between benchmarks. Since it does not account for the differences of structure, e.g., size, or behaviour between the different benchmarks, we provide a comparison across the whole benchmark set, and then compare the results according to the three categories of benchmarks (minimally-polymorphic, polymorphic and megamorphic) that we defined in Chapter 3.

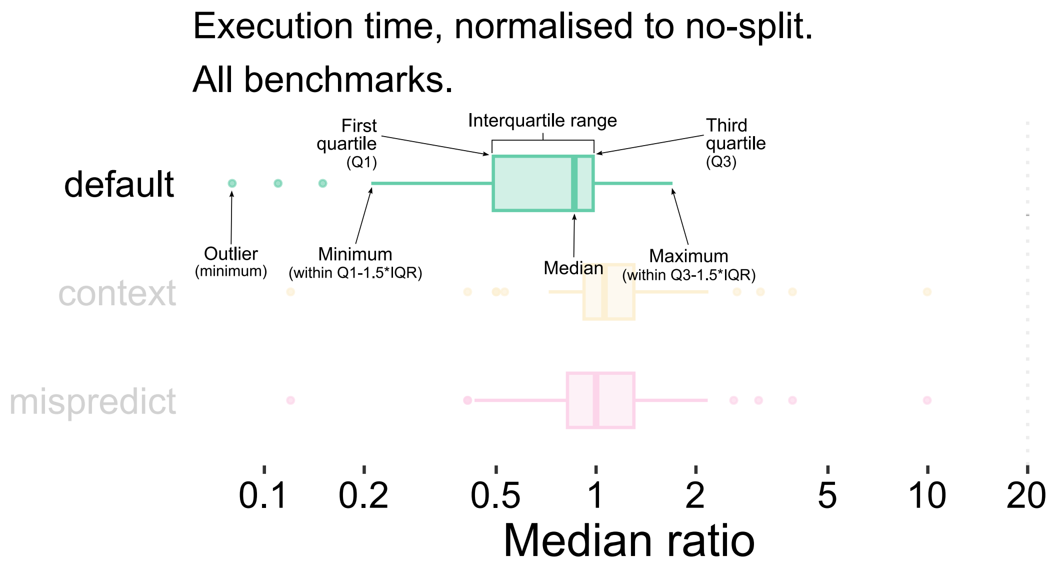


Figure 26: Execution time, warmup excluded, on a logarithmic scale, for all categories of benchmarks. The green boxplot, representing the performance of DEFAULT against NO-SPLIT, is annotated with the metrics we use to describe our results.

Figure 26 is one example of the plots we use. It compares the execution time across our whole benchmark set according to the splitting strategy used. The splitting strategies are displayed on the y-axis. The x-axis displays the median multiplicative ratio. Since we normalise our results to NO-SPLIT, this means that a strategy having a median ratio of one has the same performance as NO-SPLIT. Any ratio less than one denotes a speedup compared to NO-SPLIT. Any ratio greater than one denotes a slowdown compared to NO-SPLIT. For readability, we remove NO-SPLIT from the y-axis, and represent it as black vertical line at x = 1.

We describe the statistical distribution of our results using six different measurements within a single plot, and annotate Figure 26 with each of their names.

Median. The thickest vertical line in the boxplot corresponds to the median, which is the value separating the distribution in half, i.e., 50% of the observations sit below this median value, and the remaining 50% sit above this value. Considering our benchmark set is very diverse, using the median limits the impact of outliers to describe our results. In the case of DEFAULT, the median is 0.86, which means that half of the benchmarks ran with the DEFAULT strategy are running 13.9% faster than NO-SPLIT. It means that the remaining half of the benchmark set runs either faster than NO-SPLIT, but not more than 13.9% faster, or slower than NO-SPLIT. We can also see that MISPREDICTS performs similarly to NO-SPLIT, since its median ratio is equal to one.

First quartile (Q1) and Third quartile (Q3). The leftest vertical bar in the boxplot represents the first quartile, which means that 25% of the observations sit below its value. On the plot, it means that 25% of the observations are on the left of the vertical line representing the first quartile. For DEFAULT, this means that 25% of our benchmarks ran with the DEFAULT strategy run 50.1% faster than NO-SPLIT. The rightmost vertical bar in the boxplot represents the third quartile, which means that 75% of the observations sit below its value. On the plot, it means that 75% of the observations are on the left of the vertical line representing the third quartile. For DEFAULT, this means that 75% of our benchmarks ran with the DEFAULT strategy run at least 1.6% faster than NO-SPLIT.

Interquartile range (IQR). The first and third quartiles are useful to compute the IQR, which is the difference between the third and first quartiles. It allows to appreciate the spread of our distribution. It means that 50% of our observations are comprised between the values of Q1 and Q3. In our plot in Figure 26, it means that 50% of our observations are 1.6% faster, but not faster than 50.1%, compared to NO-SPLIT.

Minimum and maximum (without outliers). There is an horizontal line on each side of the boxplot. The bounds of these lines represent the minimum (left bound of left line) and maximum (right bound of right line) of the distribution, excluding outliers. For DEFAULT, the minimum ratio is equal to 0.25, and the maximum ratio is equal to 1.72. Any observation sitting beyond these bounds are considered to be outliers.

Minimum and maximum outliers. Several coloured dots are displayed on the left side of the boxplot. They represent the minimum outliers, i.e., the observations for which we observe the most important speedup compared to NO-SPLIT. For instance, for DEFAULT, one benchmark, ImageDemoSobel (name not displayed on the plot), is performing the fastest, running -91.9% faster than NO-SPLIT. It is represented by the leftest dot on the DEFAULT axis. Similarly, the coloured dots situated on the right side on the boxplot represent the maximum outliers, i.e., the observations that slow down the most compared to NO-SPLIT. For CONTEXT for instance, ImageDemoConv is running 9.96× slower than NO-SPLIT.

The boxplots we use in this Chapter are built on summarised data, since the execution of a benchmark is characterised by many different aspects, i.e., number of runs, number of iterations, splitting strategy used. In an attempt to limit measurement noise and limit the impact of outliers, we build an “idealised” run for each of the benchmarks by aggregating the values from several runs. We process the benchmark data as follows to build these boxplots, here re-using the example of execution time pictured in Figure 26. The same process is applied to allocated memory, time spent in garbage collection and compilation time, and the other performance metrics that are presented in this Chapter.

1. First, we run all the benchmarks for a given number of iterations, and monitor the time spent in each iteration. Each run is repeated five times. We compute the median execution time per iteration, per benchmark, out of these five runs. The end result represents an idealised benchmark run, lowering the impact of noise from our measurements. The benchmark Sieve, for instance, is run for 200 iterations, five times, for each of our four splitting strategies, totalling 4000 data points. We compute the median execution time per iteration, per strategy, and at the end of this step, we therefore go from 4000 data points (execution time in ms) to 800 data points (median execution time in ms).

2. Then, we compute the baseline data. We compute one median execution time per benchmark for the NO-SPLIT setting: we use this median value as a baseline iteration value for a given benchmark. This value represents an idealised iteration of the benchmark, where we use the median of all iterations, as well as discarding startup iterations when identifiable, to reduce the impact of outliers. Still following the example of Sieve, we start this step with 200 data points, i.e., one data point (median execution time in ms) per iteration, and compute the median execution time for the benchmark, ending up with one data point, 406.4 (median of median execution time in ms), representing the idealised median execution time for Sieve.

3. We then assess how different the three other splitting strategies, DEFAULT, CONTEXT and MISPREDICTS, perform in comparison to NO-SPLIT. To do so, we compute a multiplicative ratio, which we display on the x-axis of our boxplots. To obtain this ratio, we divide the median execution time of each iteration of each benchmark by the baseline median corresponding to the benchmark. For Sieve, we start this step with 800 data points, each representing the median execution time in ms of an iteration. At the end of this step, we still have 800 data points for Sieve, each representing the multiplicative ratio, i.e., how the median time spent in an iteration differs from baseline.

4. In the last step, we further aggregate our results to get an overall idea of how a benchmark performs according to the splitting strategy. We therefore compute a median of all the multiplicate ratios per benchmark and splitting setting. This is the value we use in the boxplots. In this step, we obtain four data points for each benchmark, i.e., one per splitting strategy. Sieve for instance, performs similarly to the baseline NO-SPLIT, since the ratio we obtain for DEFAULT, CONTEXT and MISPREDICTS are respectively equal to 0.96, 1 and 1. A ratio of one means that the benchmark performs similarly to NO-SPLIT.

In most of this Chapter, we present results at the splitting strategy granularity (see the y-axis on our plots), ignoring benchmarks specificities (see Figure 27a for instance), but we also detail the results according to the benchmark category (see for instance Figure 27b for benchmarks of the minimally-polymorphic category). In that case, we group data slightly differently in Step 3: we compute a median ratio per each `(Splitting setting, Benchmark type)` pair.

How to read the multiplicative ratios. A median ratio of 1 means that a given splitting strategy performs the same as the baseline NO-SPLIT. A median ratio lower than 1 means that a given strategy is performing better than the baseline. For instance, for execution time, a median ratio of 0.82 means that the strategy is performing 18% faster than the baseline. A median ratio higher than 1 means that a given strategy is performing worse than baseline. For instance, a median ratio of 1.21 means that the strategy is performing 21% slower than the baseline. In text, we usually mention the difference in terms of percentage rather than as a ratio, for better readability. We also mention the minimum and maximum values into parenthesis. In that case, since these values are often extreme, we use percentage when the difference is lower than 100%, and use ratios for any bigger difference. We use a logarithmic scale in our plots since it graphically displays the same distance of 0.5 and 2 from 1, easing the readability of our x-axis that displays multiplicative ratios.

5.2 Impact on behaviour: Overview of changes

We analysed the behaviour of TruffleRuby benchmarks in Chapter 3 and show that almost all polymorphism is eliminated by the optimisations in place, though splitting seems to be triggered too often. Since we developed Context-Guided Splitting in an attempt to address this issue, we examine in this Section the impact of Context-Guided Splitting, with handling mispredictions (MISPREDICTS) and without (CONTEXT), on the same aspects of program's behaviour, i.e. the degree of polymorphism, and the state of the lookup caches (see Section 3.1.3 for more details). We show that both approaches fully eliminate megamorphism, and only leave a marginal amount of polymorphic call-sites in the benchmark set. We also investigate whether either of the Context-Guided Splitting approaches has an effect on the number of monomorphic caches being split. While we notice a decrease in the volume of splits, further examined in Section 5.4.1, the tendency to split monomorphic call-sites persists; we discuss the reasons for such a result in Section

5.2.2.

We use a custom, instrumented implementation of TruffleRuby to monitor the lookup cache behaviour in our benchmark set. The details of this instrumentation are listed in Chapter 3. This same instrumentation is re-used in this Chapter to analyse the impact of Context-Guided Splitting on behaviour.

5.2.1 Impact on lookup caches' state

The goal of Context-Guided Splitting is to avoid splitting deemed unnecessary, while keeping splitting as effective as before in terms of monomorphisation. Tables 8 and 9 show the impact of Context-Guided Splitting on the monomorphisation of polymorphic and megamorphic calls, after target polymorphism is eliminated. Similarly to the Tables we discussed in Chapter 3, in Section 3.2.2, these tables show, in the Number of Calls column, the number of polymorphic and megamorphic calls in the program, after the elimination of target polymorphism. The "Change after splitting" column shows the impact of splitting on these polymorphic and megamorphic calls. A 100% decrease means that all remaining polymorphism was eliminated by splitting. For brevity, we only include megamorphic and polymorphic benchmarks in the Tables of this Section, respectively coloured dark and light grey. The full tables, featuring minimally-polymorphic benchmarks (left in white), are available in Appendix B.2.

Table 8 shows these results for the CONTEXT strategy, and Table 9 for the MISPREDICTS strategy. We see that both Context-Guided Splitting strategies completely monomorphise the remaining megamorphic calls, i.e., once target duplication is eliminated, and splitting is applied, there are no megamorphic call-sites remaining. However, not all polymorphism is eliminated by Context-Guided Splitting. We can see that some polymorphic calls remain in BlogRails, ERubiRails and Sinatra. Such results are explained by the way Context-Guided Splitting deal with mispredictions. In CONTEXT, when a misprediction occurs, i.e., when a shared

Table 8: Impact of the CONTEXT splitting strategy on the monomorphisation of polymorphic and megamorphic calls. All remaining megamorphism is eliminated by CONTEXT. A marginal number of polymorphic calls remain in BlogRails, ERubiRails and Sinatra.

	Number of calls		**Change after splitting**		**Number of splits**
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**	
BlogRails	799,623	190	-98.5%	-100%	1198
ChunkyCanvas*	228	0	-100%	0%	30
ChunkyColor*	228	0	-100%	0%	30
ChunkyDec	228	0	-100%	0%	30
ERubiRails	211,799	160	-99.8%	-100%	950
HexaPdfSmall	1,713,811	2,066	-100%	-100%	605
Lee	12,184	0	-100%	0%	381
LiquidCartParse	677	0	-100%	0%	71
LiquidCartRender	2,534	0	-100%	0%	133
LiquidMiddleware	650	0	-100%	0%	76
LiquidParseAll	1,079	0	-100%	0%	78
LiquidRenderBibs	32,807	0	-100%	0%	113
MailBench	36,764	0	-100%	0%	490
PsdColor	7,724	0	-100%	0%	202
PsdCompose*	7,724	0	-100%	0%	202
PsdImage*	7,730	0	-100%	0%	202
PsdUtil*	7,720	0	-100%	0%	202
Sinatra	2,274	220	-94.6%	-100%	170
ADConvert	12,127	0	-100%	0%	128
ADLoadFile	10,104	0	-100%	0%	105
DeltaBlue	634	0	-100%	0%	52
OptCarrot	18,167	0	-100%	0%	55
RedBlack	8,015,482	0	-100%	0%	40

method contains one or more polymorphic call-sites due to a context misprediction, the method remains shared and unsplit, and therefore, the polymorphism remains. In MISPREDICTS, when a misprediction occurs, the shared method is unvalidated and the DEFAULT splitting heuristics is applied on this method for the next calls. In that case, the remaining polymorphism is likely explained by the fact that these methods were not called after the misprediction occurred, and therefore could not be monomorphised by the default splitting strategy on the next call.

5.2.2 Focus on splitting transitions

Context-Guided Splitting monomorphises most of the polymorphic and megamorphic call-sites, similarly to the DEFAULT strategy. In Section 3.2.3, we showed that DEFAULT mostly splits monomorphic call-sites, which we consider to be over-splitting. We track whether we observe the same behaviour for CONTEXT and

Table 9: Impact of the MISPREDICTS splitting strategy on the monomorphisation of polymorphic and megamorphic calls. All remaining megamorphism is eliminated by MISPREDICTS. Like CONTEXT, a marginal number of polymorphic calls remain in BlogRails, ERubiRails and Sinatra. The number of splits is higher than CONTEXT since a context misprediction will revert to the default splitting strategy.

	Number of calls		**Change after splitting**		**Number**
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**	**of splits**
BlogRails	801,093	190	-98.7%	-100%	3294
ChunkyCanvas*	228	0	-100%	0%	38
ChunkyColor*	228	0	-100%	0%	38
ChunkyDec	228	0	-100%	0%	38
ERubiRails	211,372	160	-99.8%	-100%	2214
HexaPdfSmall	1,713,777	2,066	-100%	-100%	1134
Lee	12,110	0	-100%	0%	658
LiquidCartParse	677	0	-100%	0%	128
LiquidCartRender	2,534	0	-100%	0%	260
LiquidMiddleware	650	0	-100%	0%	111
LiquidParseAll	1,079	0	-100%	0%	162
LiquidRenderBibs	31,492	0	-100%	0%	205
MailBench	36,210	0	-100%	0%	909
PsychLoad	116,055	0	-100%	0%	694
Sinatra	2,163	220	-99.5%	-100%	345
ADConvert	12,127	0	-100%	0%	449
ADLoadFile	10,104	0	-100%	0%	363
DeltaBlue	634	0	-100%	0%	74
OptCarrot	6,452	0	-100%	0%	101
RedBlack	8,167,013	0	-100%	0%	47

MISPREDICTS and check whether they limit oversplitting. The results are displayed in Table 10 (CONTEXT) and Table 11 (MISPREDICTS). The first column denotes the number of splits. The second column contains the percentage of splits that led to different lookup cache entries, i.e., when a method gets split and its lookup caches contains a different set of entries before and after the split. The third column contains the percentage of splits that led to unchanged lookup cache entries, which means the split was unnecessary.

We see in Table 10 that 17% (min. 8%, max. 42%) of splits lead call-sites to have different lookup cache entries, and consequently 83% (min. 58%, max. 92%) of splits lead to unchanged lookup cache entries, i.e., oversplitting. Among the polymorphic and megamorphic benchmarks depicted in this table, the RedBlack benchmark is displaying the least oversplitting overall, with 40% of splits leading to different lookup cache's entries. As we expected, CONTEXT has a positive impact of this kind of oversplitting, i.e., monomorphising monomorphic call sites:

Table 10: Splitting transitions for the CONTEXT strategy. While the majority of splitting is performed on monomorphic call-sites, CONTEXT succeeds at limiting this kind of oversplitting compared to the other strategies, with 17% (min. 8%, max. 42%) caches changed.

		Cache entries after splitting (% of total number of splits)	
Benchmark	**Number of splits**	**Different**	**Same**
BlogRails	1198	24%	76%
ChunkyCanvas*	30	10%	90%
ChunkyColor*	30	10%	90%
ChunkyDec	30	10%	90%
ERubiRails	950	23%	77%
HexaPdfSmall	605	20%	80%
Lee	381	19%	81%
LiquidCartParse	71	8%	92%
LiquidCartRender	133	11%	89%
LiquidMiddleware	76	13%	87%
LiquidParseAll	78	9%	91%
LiquidRenderBibs	113	17%	83%
MailBench	490	18%	82%
PsdColor	202	17%	83%
PsdCompose*	202	17%	83%
PsdImage*	202	17%	83%
PsdUtil*	202	17%	83%
Sinatra	170	19%	81%
ADConvert	128	25%	75%
ADLoadFile	105	22%	78%
DeltaBlue	52	42%	58%
OptCarrot	55	16%	84%
RedBlack	40	40%	60%

DEFAULT and MISPREDICTS (see Table 11) lead to respectively 90.5% (min. 60%, max. 94%) and 93% (min. 66%, max. 96%), which means that a higher number of splits under these two strategies applies on monomorphic call-sites. CONTEXT shares specialised methods between different call-sites, even when these methods get polymorphic after a misprediction, therefore limiting the total amount of splits. The better performance of CONTEXT in terms of oversplitting can be explained by this strategy: many of the avoided splits would have been triggered on monomorphic methods.

The impact of MISPREDICTS on oversplitting is not as positive, as shown in Table 11. As already mentioned in Section 5.2.1, and further discussed in Section 5.4.1, MISPREDICTS triggers more splits than CONTEXT, since it reverts to the DEFAULT strategy in case of a misprediction, which leads to oversplitting as shown

Table 11: Splitting transitions for the MISPREDICTS strategy. Since MISPREDICTS relies on DEFAULT and leads to a higher number of splits than CONTEXT, it has a lower impact on limiting the monomorphisation of already monomorphic call-sites.

		Cache entries after splitting (% of total number of splits)	
Benchmark	**Number of splits**	**Different**	**Same**
BlogRails	3294	9%	91%
ChunkyCanvas*	38	8%	92%
ChunkyColor*	38	8%	92%
ChunkyDec	38	8%	92%
ERubiRails	2214	10%	90%
HexaPdfSmall	1134	11%	89%
Lee	658	11%	89%
LiquidCartParse	128	5%	95%
LiquidCartRender	260	6%	94%
LiquidMiddleware	111	9%	91%
LiquidParseAll	162	4%	96%
LiquidRenderBibs	205	9%	91%
MailBench	909	10%	90%
PsychLoad	694	11%	89%
Sinatra	345	9%	90%
ADConvert	449	7%	93%
ADLoadFile	363	6%	94%
DeltaBlue	74	30%	70%
OptCarrot	101	9%	91%
RedBlack	47	34%	66%

in Section 5. Therefore, we observe both a higher number of splits compared to CONTEXT, and a higher volume of oversplitting.

This kind of oversplitting may be further limited if we would share a set of specialised methods rather than one specialised method as it is the case currently in CONTEXT and MISPREDICTS. We further discuss this approach as part of Future Work, in Chapter 6.

5.3 Impact on execution time

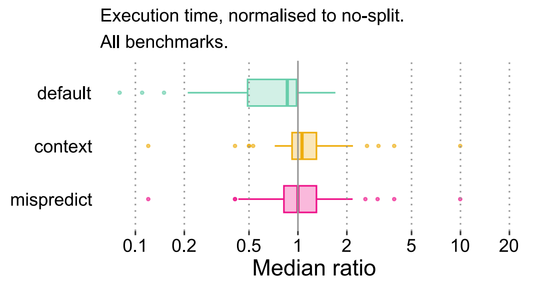
We showed in Section 3.3.2 that DEFAULT, the standard splitting heuristics in TruffleRuby, has a positive impact on execution time. We investigate in this section whether we can observe a similar impact with CONTEXT and MISPREDICTS. In this Section, we distinguish between peak and startup performance. A program usually requires some time at the beginning of the execution to warm up, e.g., to initialise objects, optimise the program and exit interpreter mode. This is called the startup phase, during which the execution time is generally higher, since the executed code has not been properly optimised yet. The performance observed is also generally unstable since run-time profiling and many compilation operations are performed. Ideally, the startup phase should only represent a small fraction of the total execution time of a program. If a program exits its startup phase, we consider it then reaches its peak performance phase: most of its code has been optimised, and the time spent per iteration is stable.

In this Section, we first analyse peak performance in our benchmark set, by manually discarding startup-related iterations. We then explain in Section 5.3.2 how we determine the startup thresholds, i.e., whether –and when– a benchmark exits startup phase, and assess the impact of splitting on startup performance.

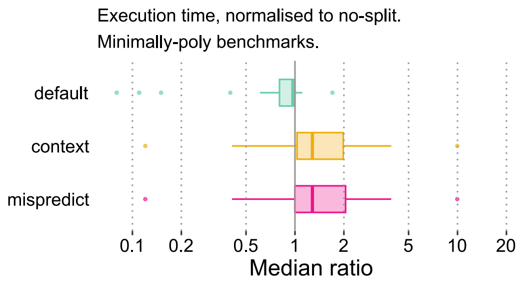
5.3.1 Peak performance

We measure the impact of the four splitting strategies on execution time, on peak performance, i.e., once the execution time of the benchmark stabilises. Figure 27 gathers these results, first across the whole benchmark set (see Figure 27a), and then with a detailed view on the benchmark category: the minimally-polymorphic benchmarks are shown in Figure 27b, the polymorphic benchmarks, in Figure 27c, and the megamorphic benchmarks in Figure 27d.

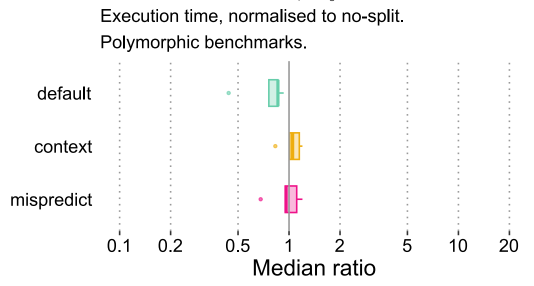
We observe in Figure 27a that the DEFAULT splitting heuristics, coloured in green, is the only strategy bringing an overall median speed-up, of 13.9% compared to NO-SPLIT (min. -91.9%, max. 70.1%). The Context-Guided Splitting approaches either lead to a slowdown, with CONTEXT slowing down execution time by 6.3% (min. -87.7%, max. 9.96×), or do not impact performance, like MISPREDICTS, which performs similarly to a case when splitting is disabled (min. -87.7%, max. 9.96×). The detailed ranges are gathered in Table 22 in Appendix B.4.



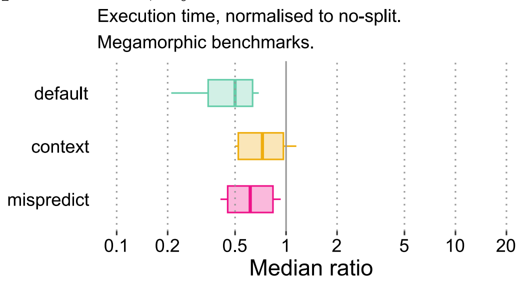
(a) All categories of benchmarks taken together. DEFAULT is the setting reducing execution time the most, by 13.9%.



(b) Minimally-polymorphic benchmarks. The DEFAULT heuristics marginally speeds up execution, by 2.6%.



(c) Polymorphic benchmarks. DEFAULT is still the most impactful approach with a speed-up of 14.4%.



(d) Megamorphic benchmarks. MISPREDICTS gets closer in performance to DEFAULT with a median speed-up of 38.6%, versus 49.9% for DEFAULT.

Figure 27: Execution time, warmup excluded, on a logarithmic scale, for all categories of benchmarks. Lower is better.

Since the benchmarks have different a optimisation potential, and that it is likely that the more polymorphic a benchmark is, the more splitting will improve performance, we now focus on performance for each benchmark category. While DEFAULT leads to a marginal speed-up of 2.6% (min. -91.9%, max. 70.1%) for minimally-polymorphic benchmarks, we observe more important speedups when the degree of polymorphism of the benchmarks increases. The same behaviour can be observed for the Context-Guided Splitting approaches. MISPREDICTS outperforms CONTEXT for polymorphic benchmarks, speeding-up execution time by 3.6% (min. -31.6%, max. 19.6%), while CONTEXT slows down the execution by 5.1% (min. -16.6%, max. 20%).

We observe the same behaviour the megamorphic benchmark category. In that case, the performance of MISPREDICTS gets closer to DEFAULT, with a speed-up of 38.6% (min. -59.2%, max. -7.2%) compared to 49.9% (min. -79%, max. -30.9%) for DEFAULT. These results correlate with the assumption that the most polymorphic benchmarks will benefit the most from splitting, since there are many call-sites to monomorphise, therefore improving performance compared to a case where call-sites would remain megamorphic. We assume CONTEXT performs generally slower since it may dispatch to split method that are not specialised to the current context in case of a misprediction. It may therefore hamper optimisation potential and impact performance.

5.3.2 Impact on startup performance

Most programs tend to experience a warm-up–or startup–phase, representing the time needed for the system to reach a steady-state of performance. We assume splitting may have a positive impact on startup performance: it helps dispatching to methods that have already been profiled and specialised, therefore presumably saving the cost of splitting and specialisation of following split methods. Splitting is performed during interpretation, and startup is dominated by interpreter performance. Therefore, while we do not expect a critical improvement of peak performance, we expect splitting might be beneficial for startup, especially for polymorphic benchmarks with a long startup phase. Limiting the time spent in startup may be desirable, for instance, in the case of an application executed on a pay-per-use platform, or when an application is deployed frequently.

Since startup performance is generally unstable, displaying a lot of peaks in execution time, and may not have the same duration depending on the splitting strategy used, we therefore display the results of this Section using different plot visualisations to account for all these aspects.

Changepoint analysis to identify a startup phase. We use the changepoint analysis, similarly as Barrett et al. (2017), to roughly identify whether, and when, a benchmark reaches a steady state of performance. We select slightly looser criteria, aiming at identifying one changepoint at most per execution, and therefore rely on the AMOC (*At Most One Change*, which identifies a single changepoint) algorithm rather than PELT.

Figure 28 shows a graphical representation of the switch from startup phase to a steady state of performance. The first green horizontal line represents the observed mean, as well as the start and end iterations of the startup phase, for the BlogRails benchmark. We use the thresholds gathered using this algorithm to identify a potential startup phase for each of our benchmarks with each of the splitting strategies.

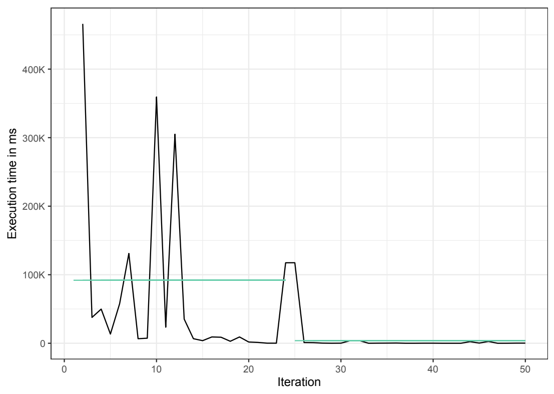
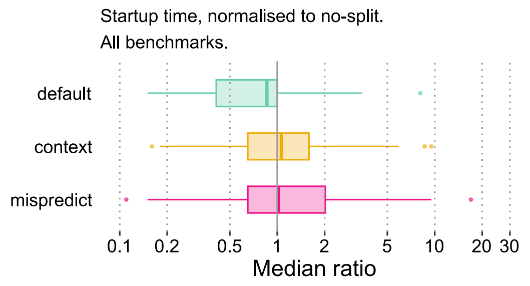


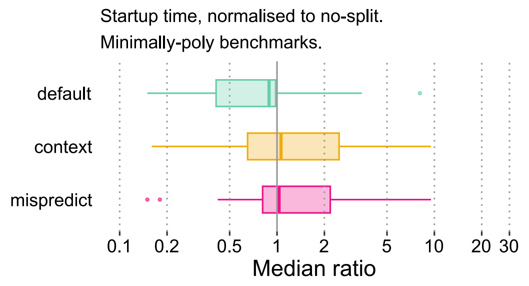
Figure 28: We use the AMOC algorithm to determine if a benchmark reaches a steady state of performance. Here, we observe that BlogRails exits the startup phase on iteration 24.

As already demonstrated by Barrett et al. (2017), some benchmarks may never reach a steady-state of performance. We also observed such behaviour in our benchmark set for several of the benchmarks, for the amount of iterations we chose. We include these benchmarks in the results of this Section, since we can consider they never stabilise, and therefore we do not discard any iteration of their runs. They however have been discarded from the other sections since we build our other observations on peak performance. For brevity, we only include polymorphic and megamorphic benchmarks in Figures 30 and 32, but the full results are available in Appendix B.3.

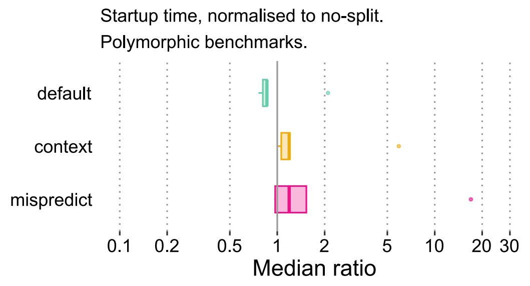
Median time per startup iteration. We first focus on the median time spent per startup iteration. We compute the median ratio compared to NO-SPLIT, similarly as in the other Sections, and show the aggregated results in Figure 29. To build these plots, we only keep the startup iterations, as identified by the changepoint analysis, and build our statistical summary on this subset of iterations.



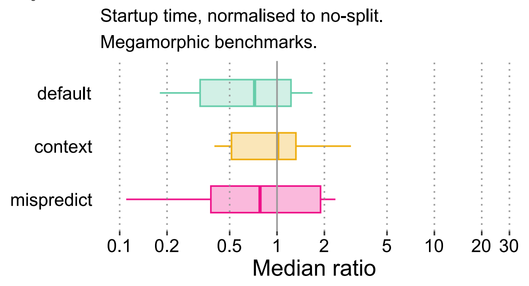
(a) All categories of benchmarks taken together. DEFAULT is the only setting reducing startup time, by 13.6%.



(b) Minimally-polymorphic benchmarks. DEFAULT decreases the time spent per startup iteration, by 11.1%. The Context-Guided Splitting strategies perform similarly to NO-SPLIT.



(c) Polymorphic benchmarks. DEFAULT is the most impactful approach with a speed-up of 13.9%.



(d) Megamorphic benchmarks. DEFAULT is the most impactful setting, with a median speed-up of 28%, followed by MISPREDICTS, with a speedup of 22%.

Figure 29: Startup time, on a logarithmic scale, for all categories of benchmarks. Lower is better.

We observe a similar trend to the peak performance results. The DEFAULT splitting strategy is the most beneficial overall, leading to a faster time per startup iteration for all benchmark categories. We can see in Figure 29a it decreases the time per startup iteration by 13.6% overall, and benefits the most to megamorphic benchmarks, which spend 28% less time in startup compared to NO-SPLIT (see Figure 29d). The Context-Guided Splitting heuristics are less successful: overall, CONTEXT has a limited impact on startup performance, with a median startup time close to NO-SPLIT. Similarly, MISPREDICTS is not impacting startup performance for minimally-polymorphic benchmarks. However, it leads to a 19.1% slowdown for polymorphic benchmarks, but significantly decreases the startup time of megamorphic benchmarks, by 22%. MISPREDICTS is also the strategy bringing the biggest speedup, for LiquidCartRender, a megamorphic benchmark, whose startup time got reduced by 89.4%.

Duration of the startup phase. The metric we use in the previous paragraph, i.e. the median time spent per startup iteration compared to NO-SPLIT, does not account for the impact of the different splitting strategies on the time needed to exit the startup phase, i.e., for the execution time to stabilise. It assesses how much time is spent in a startup iteration overall, and to which extent this time is different from baseline. However, the number of iterations spent in startup may be different according to the splitting strategy applied. In this Section, our goal is to measure the total cumulated time spent in startup, and check whether it is impacted by the splitting strategy.

We represent the cumulated time spent in startup with a stacked bar plot, in Figure 30, for megamorphic benchmarks only. The plot for polymorphic and minimally-polymorphic benchmarks in available in Appendix 41. The bars represent the cumulated time spent in the benchmark. The time in seconds is displayed on the x-axis. The first section of the bar, coloured in orange, represents the cumulated time spent in the startup phase, and the second part of the bar coloured in green represents cumulated the time spent in the steady state phase. The benchmarks that do not reach a steady state of performance are considered to never exit the startup phase, and are therefore depicted with only an orange bar. For instance,

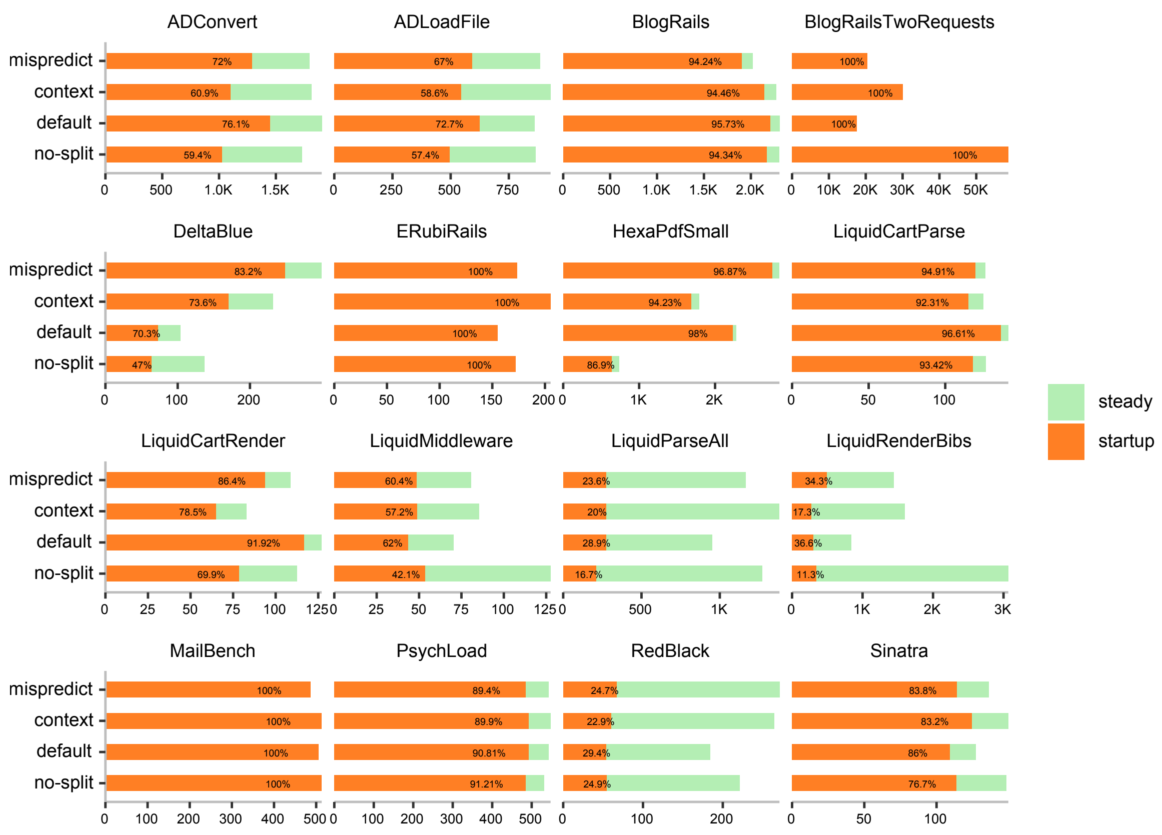
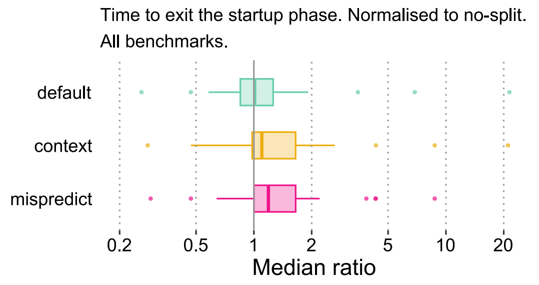


Figure 30: The four splitting strategies have varying impact on the duration of the startup phase in our benchmarks.

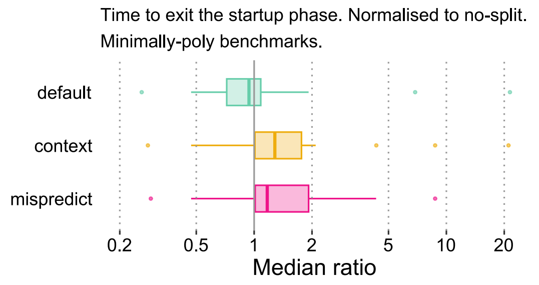
we observe that BlogRailsTwoRequests, ERubiRails and MailBench do not reach a steady state. There are four bars per benchmark, one bar corresponding to each of the four splitting strategies (y-axis). Since the different splitting strategies may have a different impact on the total duration of the startup, we also include on each bar the percentage representing how important the startup phase is relative to the total execution time.

Over the 13 megamorphic and polymorphic benchmarks reaching a steady state, NO-SPLIT reaches a steady state earlier than the other strategies for six benchmarks, like for instance, ADConvert and ADLoadFile. DEFAULT and CONTEXT are reducing the time spent in startup for three benchmarks each, LiquidMiddleware, RedBlack and Sinitra for DEFAULT, and LiquidCartParse, LiquidCartRender and

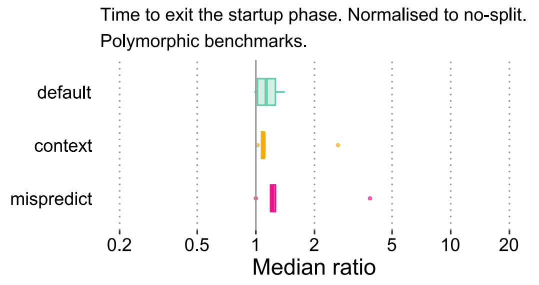
LiquidRenderBibs for CONTEXT. MISPREDICTS reduces the time spent in startup for one benchmark, BlogRails, which is one of the largest benchmarks of our set. The strategy leading to the most important reduction in startup time is CONTEXT: it reduces the time spent in startup of LiquidRenderBibs by 21.4%, compared to NO-SPLIT. On the other hand, MISPREDICTS is the strategy leading to the biggest slowdown observed in megamorphic benchmarks, for HexaPdfSmall, exiting of the startup phase 4.3× later than NO-SPLIT.



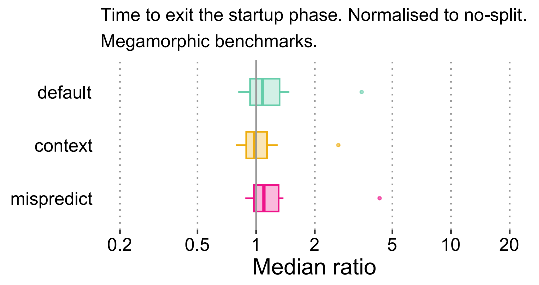
(a) All categories of benchmarks taken together. DEFAULT has the least impact on the median time needed to exit the startup phase, overall exiting 1% later than NO-SPLIT.



(b) Minimally-polymorphic benchmarks. DEFAULT is the only beneficial strategy, exiting the startup phase 5.9% earlier than NO-SPLIT.



(c) Polymorphic benchmarks. CONTEXT is the strategy exiting the startup phase earlier than the other strategies, though exiting 9.8% later than NO-SPLIT



(d) Megamorphic benchmarks. CONTEXT is the most impactful setting, with a median exit 2% than NO-SPLIT.

Figure 31: Impact of the splitting strategies on the time needed to exit the startup phase, on a logarithmic scale, for all categories of benchmarks. Lower is better.

Figure 31 provides a different perspective on these results, where we use a boxplot representation to compare the impact of the startup duration per benchmark category rather than on a per-benchmark basis. We observe that, compared to NO-SPLIT, the other splitting strategies are generally making the startup phase last longer. However, Figure 27, in the previous Section 5.3.1 where we focus on peak performance, shows that they usually decrease the overall duration of the benchmarks. If we focus on the impact on the different categories on benchmarks (Figures 31b to 31d), we see that DEFAULT is generally reducing the time spent in startup for minimally-polymorphic benchmarks by 5.9%. Polymorphic benchmarks usually benefit more from having splitting disabled. Megamorphic benchmarks (Figure 31d) see their startup time decreased by 2% with the CONTEXT splitting strategy. Overall, splitting has a limited impact on the total time spent in the startup phase. On a per-benchmark basis, it tends to benefit more to benchmarks having their execution time dominated by startup, fitting in the megamorphic category.

Focus on the first iteration. In Section 5.3.2, we investigate the impact of the splitting strategies on the time per iteration during startup. We follow up on this approach, and focus now on a more extreme case: the time spent in the first iteration of a benchmark. This metric is relevant for short running applications, in particular in the context of FaaS or SaaS, where it is likely an application, or a single method, is executed once, therefore not running long enough exit the startup phase. This is especially critical in the case of a pay-per-use policy. For this aspect, we ran the first iteration of our benchmarks, 50 or 100 times depending on the time required for a run, and compute the median of these runs to have one data point per iteration. We represent the median time spent in the first iteration of our polymorphic and megamorphic benchmarks in Figure 32. The plot for minimally-polymorphic benchmarks is available in Appendix 42. For readability, instead of a ratio, we chose to represent the difference of performance against NO-SPLIT, in milliseconds. Any negative result denotes a better performance than NO-SPLIT for the first iteration.

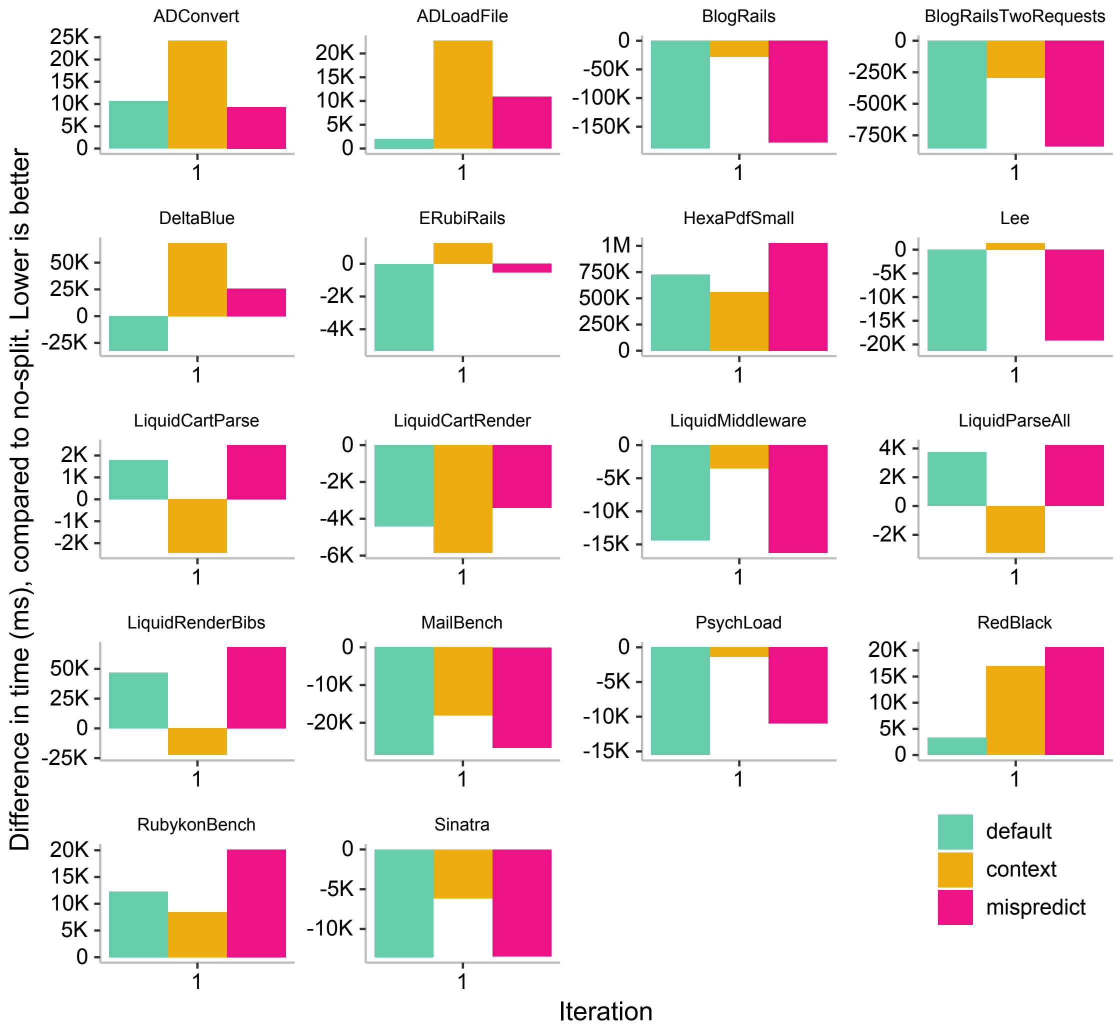


Figure 32: Polymorphic and Megamorphic benchmarks only. Impact of the splitting strategies on the time spent in the first iteration of the benchmarks. DEFAULT is overall leading to the fastest first iteration on 8 benchmarks. Disabling splitting is beneficial for the first iteration of five benchmarks. CONTEXT is performing best for the four remaining benchmarks.

Among the polymorphic and megamorphic benchmarks, we observe that DEFAULT is the best performing strategy for the first iteration. NO-SPLIT is performing best for five benchmarks (ADConvert, ADLoadFile, HexaPdfSmall, RedBlack and Rubykon), i.e., it displays the shortest execution time of all splitting strategies. Of the remaining strategies, CONTEXT is the best performing for four benchmarks. MISPREDICTS displays a more mixed performance, leading to better performance for LiquidMiddleware. It yields a significant decrease in the time spent in the first iteration for Sinatra, and the two BlogRails benchmarks, but is often outperformed by DEFAULT on these benchmarks.

Overall, NO-SPLIT is a valid strategy if the startup phase is the main performance aspect one wants to improve. While it may not lead to the generation of highly efficient compiled code, NO-SPLIT has overall a low impact on startup performance. However, this observation mostly applies to benchmarks having low polymorphism. More realistic benchmarks, representing real-world workload, are likely to display a higher level of polymorphism as we observed in Chapter 3. We showed in this Section that many of these polymorphic and megamorphic benchmarks see their startup performance improved by splitting. While DEFAULT brings an overall benefit on our set, we saw, looking on a per-benchmark basis, that Context-Guided Splitting benefits the most to the real-world benchmarks we use, e.g., BlogsRails, LiquidCartRender, LiquidCartParse. Compared to MISPREDICTS, CONTEXT especially leads to better startup results, mostly because it ignores context mispredictions and therefore is more lightweight than MISPREDICTS.

5.4 Impact on memory usage

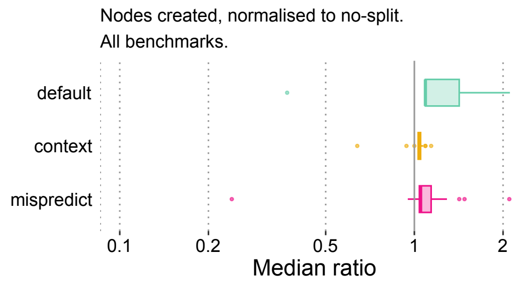
Since Context-Guided Splitting aims at reducing the amount of splits, we expect it to have a positive impact on memory usage, as fewer methods should be cloned. Since this should directly impact the size of the AST, we first compare the total number of AST nodes generated at run time. We then investigate whether the splitting strategies have an impact on garbage collection-centred metrics, such as the volume of allocated memory and the time spent in garbage collection. Since AST nodes are part of the allocated memory, we check how the two metrics interact. Our analysis leaves out some memory-related aspects, notably the lifetime behaviour, whether it differs according to the benchmark category, and how it may be impacted by the splitting strategy.

We use the G1 Garbage Collector as our default garbage collector. It is a Stop-The-World, parallel, incremental and generational garbage collector. The heap is divided in many fixed-sized regions, themselves belonging to one of two virtual spaces: the young space or the old space. The young space is itself divided into two, the Eden regions, containing newly allocated, younger objects and the Survivor regions, containing objects that have already been reclaimed. Objects from the Survivor space are moved to the Old space if they survive, i.e., are still in use, long enough. The moves from Eden to Survivor spaces are triggered by "minor" garbage collections, called scavenges. Once the Old space is full, a full, or "major" GC is triggered, to reclaim now-unused objects in all spaces.

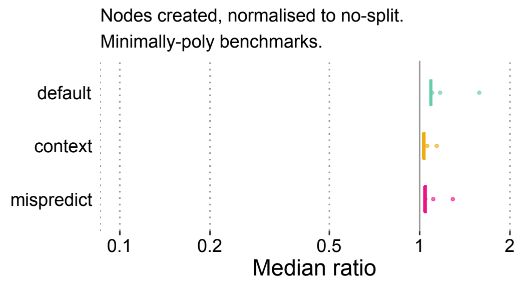
5.4.1 The size of the AST decreases

Figure 33 displays the median volume of nodes created at run time, according to the benchmark category and the splitting approach chosen. We gather this data by using the instrumentation counters `totalCreatedNodeCount` and `splitNodeCount` provided by Graal. The first counter tracks the volume of nodes created outside of any splitting action: the first time a method is called, its call-target, i.e. the object representing its target implementation, is created. The counter is incremented by the volume of nodes constituting the AST of this method. The second counter is incremented in the same fashion, but only for methods that are split. We add the two counters together to approximate the total number of nodes created at run time.

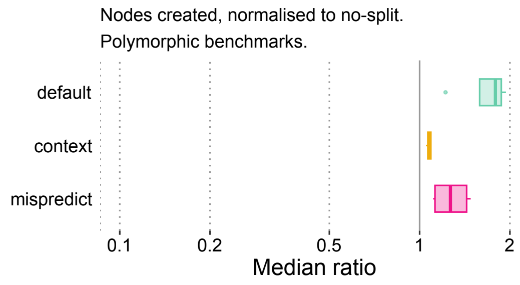
We show the overall impact of the different splitting approaches on the volume of nodes created in Figure 33a. Enabling splitting naturally increases the number of created nodes since it duplicates methods. Therefore, we observe a 9.4% increase of AST nodes for DEFAULT (min. -63.1%, max. 2.11×), and a respective 3.6% (min.



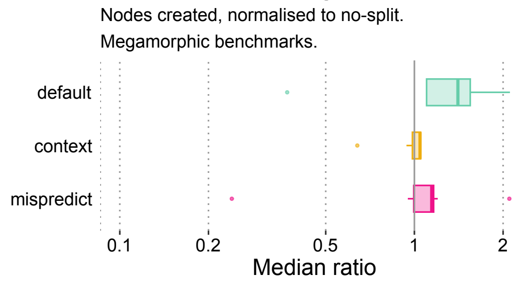
(a) All categories of benchmarks taken together. Splitting increases the volume of nodes created since it duplicates methods.



(b) Minimally-polymorphic benchmarks. Both Context-Guided Splitting approaches perform similarly and create fewer nodes than the DEFAULT splitting heuristics.



(c) Polymorphic benchmarks. The performance gap widens when the degree of polymorphism increases. CONTEXT performs best with a 8% increase of the volume of nodes created overall.



(d) Megamorphic benchmarks. CONTEXT still dominates compared to DEFAULT and MISPREDICTS.

Figure 33: Number of nodes created, on a logarithmic scale, for all categories of benchmarks. Lower is better.

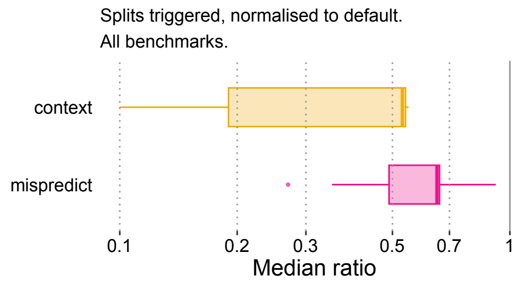
-35.6%, max. 14%) and 4.6% (min. -76.4%, max. 2.1×) increase for CONTEXT and MISPREDICTS. The CONTEXT approach generates 3.4% fewer nodes than NO-SPLIT for the minimally-polymorphic benchmarks (Figure 33b), 8% more for the polymorphic ones (Figure 33c), and 4.3% more nodes for the megamorphic benchmarks (Figure 33d).

We can put these results in perspective by comparing the performance of CONTEXT with DEFAULT as a baseline, which is the splitting approach generating the most nodes. For minimally-polymorphic benchmarks, CONTEXT performs slightly better with 5.3% fewer nodes generated compared to DEFAULT. This gap widens when the degree of polymorphism increases, and CONTEXT helps generating 39.6% fewer nodes for polymorphic benchmarks and 25.3% fewer nodes for megamorphic benchmarks. This decrease can be explained by the fact that CONTEXT limits splitting and rather reuses cloned methods, which in turn limits the overall size of the program representation.

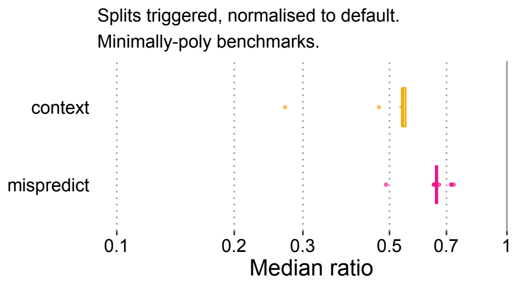
While MISPREDICTS generates fewer nodes than DEFAULT, we observe it does not perform as good as CONTEXT. This also correlates to the number of splits, described below, as the MISPREDICTS strategy will revert to a normal splitting strategy when a context misprediction occurs, therefore leading to the generation of more method clones. We further investigate the correlation between the volume of nodes created and the amount of splitting later in this section.

Considering that the number of nodes created strongly correlates with the number of splits, we also investigate how the latter is influenced by the splitting strategies. Figure 34 shows the number of splits realised per benchmark category and splitting approach. In this plot, the baseline is DEFAULT, since no split occurs in the NO-SPLIT strategy.

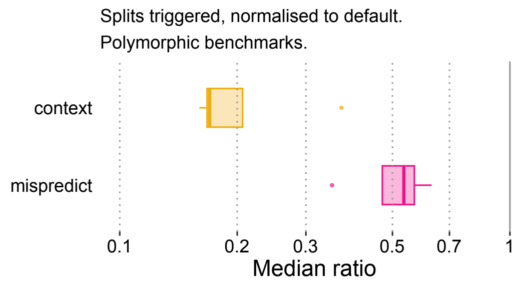
The more polymorphic the benchmarks are, the more splitting occurs, since the amount of call-sites that require splitting is more important. All benchmarks considered, we can see that CONTEXT triggers 46.7% (min. -89.6%, max. -44.5%) less splits, and MISPREDICTS 34.5% less (min. -72.9%, max. -8.4%). We notice that much less splitting is triggered for both Context-Guided Splitting approaches: for polymorphic benchmarks, CONTEXT generates 83% fewer splits than DEFAULT, and 81.6% fewer for the megamorphic benchmarks. With Context-Guided Splitting favouring method reuse over splitting, this decrease is expected. The amount of splitting with MISPREDICTS remains lower than the DEFAULT splitting setting, but much higher than with CONTEXT, with respectively 46.2% splits for the polymorphic benchmarks and 58.6% for the megamorphic ones. As mentioned earlier, this is caused by the handling of mispredictions. Indeed, when a context



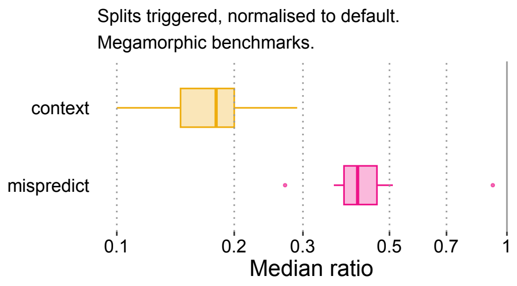
(a) All categories of benchmarks taken together. CONTEXT triggers the fewest splits overall.



(b) Minimally-polymorphic benchmarks. Both Context-Guided Splitting approaches trigger fewer splits than DEFAULT, by at least 34.3%.



(c) Polymorphic benchmarks. Notably, CONTEXT reduces the amount of splits by 83%.



(d) Megamorphic benchmarks. CONTEXT reduces the amount of splits by 81.6%.

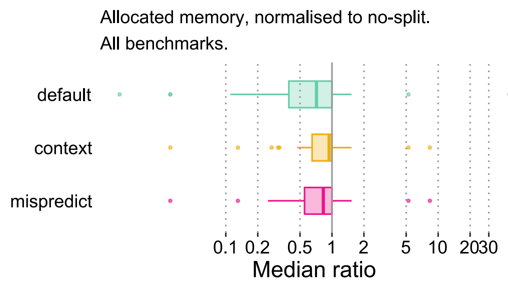
Figure 34: Number of splits triggered, on a logarithmic scale, for all categories of benchmarks. Lower is better.

misprediction occurs, we invalidate the pair `(Context, Method)` before the call is executed, and perform a regular split instead. The higher volume of splits we observe in Figure 34 for MISPREDICTS therefore correlates with the volume of mispredictions detected at run time.

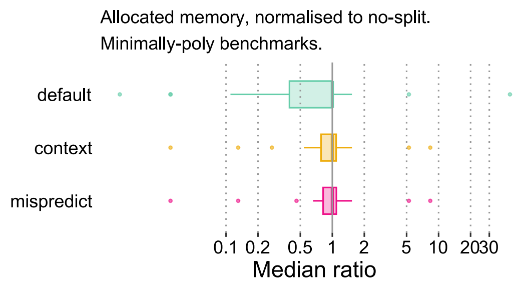
5.4.2 Allocated memory

We expect that a decrease in the volume of created nodes may improve the total allocated memory. We use the library call `getCurrentThreadAllocatedBytes()` of the `ThreadMXBean` Java interface to fetch the volume of allocated bytes of the current thread before and after each iteration of the benchmark. We subtract the two to obtain the volume of allocated memory per iteration. We show in

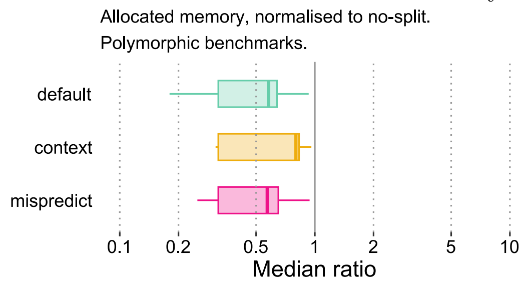
Figure 35 how the volume of allocated memory is influenced by the splitting strategy, similar to what we partially discussed in Section 3.3.2 for DEFAULT and NO-SPLIT. Similarly to Section 5.4.1, we first display the overall results across the whole benchmark set (Figure 35a), and then take a detailed view according to the benchmark category (Figures 35b, 35c and 35d).



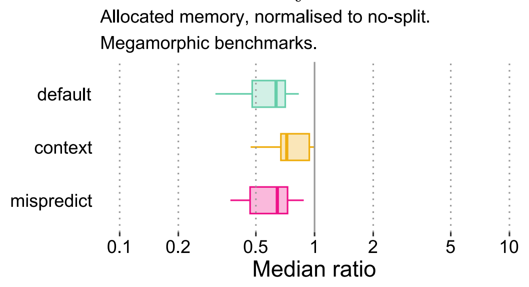
(a) All categories of benchmarks taken together. All splitting approaches seem beneficial to the volume of allocated memory.



(b) Minimally-polymorphic benchmarks. Splitting only marginally improves the volume of allocated memory.



(c) Polymorphic benchmarks. DEFAULT and MISPREDICTS decrease the volume of allocated memory similarly.



(d) Megamorphic benchmarks. The DEFAULT setting brings the most decrease, by 36.3%.

Figure 35: Allocated memory used at run time, on a logarithmic scale, for all categories of benchmarks. Lower is better.

Overall, the runtime allocates less memory when splitting is enabled: 29.9% for DEFAULT (min. -100%, max. 46.91×), 6.6% for CONTEXT (min. -100%, max. 8.35×) and 21.7% for MISPREDICTS (min. -100%, max. 8.35×).[1] We described the reasons for such a decrease in Section 3.3.2 in Chapter 4: splitting uncovers optimisation potential, typically, in that case, optimisations beneficial to memory

[1]As stated in Chapter 3, we observe such a sharp decrease in allocated memory in some of our micro-benchmarks, because splitting allows e.g. escape analysis to be performed.

usage, such as scalar replacement and boxing elimination for instance.

However, none of the Context-Guided Splitting approaches outperform the DEFAULT heuristics. It may seem surprising considering that we saw in Section 5.4.1 that these approaches were much more effective at decreasing the size of the program representation. We argue it shows that the size of the program representation has a low impact on the general volume of allocated memory, especially for longer benchmarks as discussed by Larose et al. (2023).

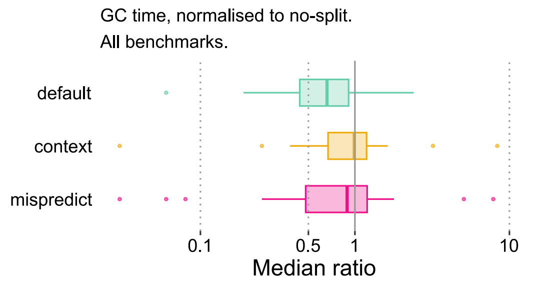
Similarly to peak performance, the more polymorphic the benchmarks are, the more splitting has a positive impact. Polymorphic benchmarks see their allocated memory respectively decreased by 42.4% (DEFAULT) and 42.7% (MISPREDICTS), and 36.3% and 35.7% for megamorphic benchmarks. CONTEXT decreases the volume of allocated memory compared to NO-SPLIT too, but does not perform as good as the other strategies, with 20.3% less allocated memory for polymorphic benchmarks, and a reduction of 28.2% of the allocated memory for megamorphic benchmarks.

5.4.3 Time spent in garbage collection

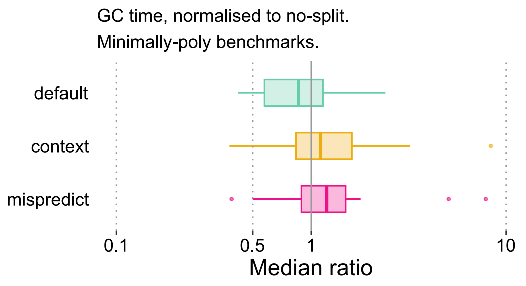
We further investigate the relation between splitting and run-time memory usage in Figure 36, which shows the impact of splitting on the time spent in garbage collection, on all benchmarks first, and for each benchmark category separately.

Overall, splitting has a positive impact on the time spent in garbage collection, with a 39.8% (min. -100%, max. 3,000,000×)[2] decrease for the MISPREDICTS strategy. The DEFAULT setting performs better with a decrease of 50% of the time spent in garbage collection (min. -100%, max. 4,000,000×). The minimally-polymorphic benchmarks only see their time spent in GC reduced by DEFAULT,

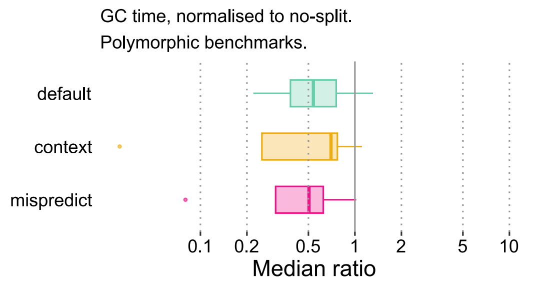
[2]As stated in the previous chapter, we observe extreme outliers when monitoring the time spent in garbage collection. We did not record any time spent in GC for Queens with NO-SPLIT for instance, but observed a median 3 ms for MISPREDICTS, leading to this 3,000,000× difference. A minimum of 100% is caused when comparing a non-zero ms baseline median to a 0ms median.



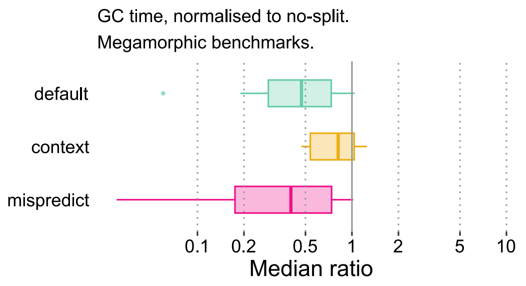
(a) All categories of benchmarks taken together. DEFAULT and MISPREDICTS significantly reduce the time spent in garbage collection.



(b) Minimally-polymorphic benchmarks. DEFAULT is the only beneficial splitting approach.



(c) Polymorphic benchmarks. MISPREDICTS significantly lowers the time spent in garbage collection, with a decrease of 51.8%.



(d) Megamorphic benchmarks. Again, MISPREDICTS significantly reduces the time spent in garbage collection, by 56.6%.

Figure 36: Time spent in garbage collection, on a logarithmic scale, for all categories of benchmarks. Lower is better

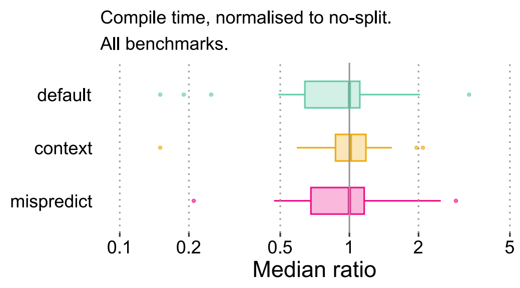
with a median decrease in time of 42.1% (min. -100%, max. 4,000,000×). However, when the degree of polymorphism of the benchmarks increases, MISPREDICTS is the best-performing strategy, reducing the time spent in garbage collection by 51.8% (min. -100%, max. 1.9%) for polymorphic benchmarks and by 56.6% (min. -96.7%, max. 2.4%) for megamorphic benchmarks.

These results seem to correlate to the decrease in the volume of allocated memory observed in Figure 35, i.e., if we observe a decrease in allocated memory, we will observe a decrease in the time spent in garbage collection.

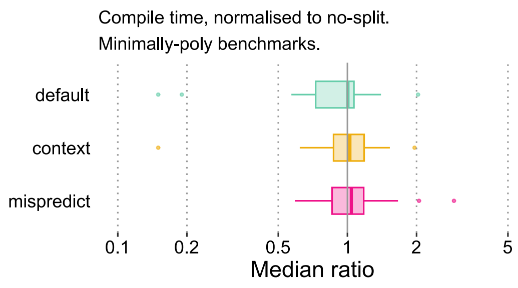
In addition, it is possible that Context-Guided Splitting performs different from DEFAULT and NO-SPLIT since it likely impacts the lifetime of objects. Indeed, we share methods as ASTs, whose nodes are allocated objects at run time. While shared methods are discarded when a misprediction occurs, “successful” shared methods may live much longer compared to the DEFAULT setting.

5.5 Impact on compilation time

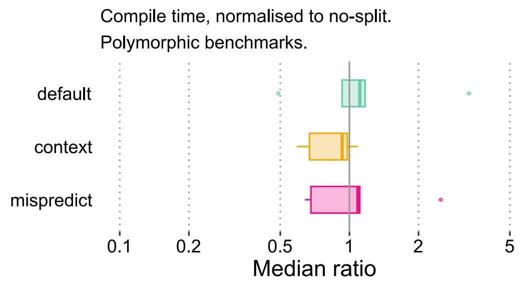
We mentioned in Section 3.3.3 that we expect Context-Guided Splitting to improve compilation time since it reduces the number of compilation units. Indeed, a split triggers the cloning of a method that may be compiled in the future, so sharing methods rather than splitting should therefore reduce the number of methods that need compiling.



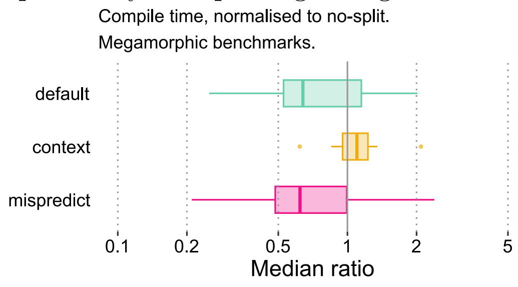
(a) All categories of benchmarks taken together. All strategies have a marginal impact on compilation time overall.



(b) Minimally-polymorphic benchmarks. Likewise, the minimally-polymorphic benchmarks do not see their compilation time impacted by the splitting stratgies.



(c) Polymorphic benchmarks. CONTEXT is still the only beneficial approach, with a 7.5% decrease.



(d) Megamorphic benchmarks. MISPREDICTS now significantly decreases the time spent compiling, by 37.5%.

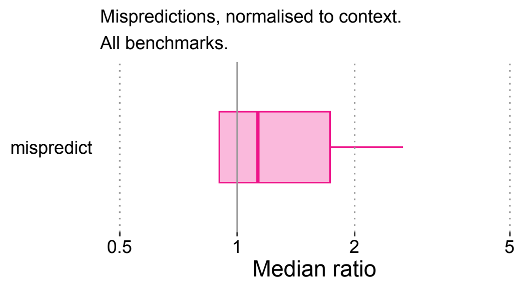
Figure 37: Run-time compilation time, on a logarithmic scale, for all categories of benchmarks. Lower is better

Following the same reasoning, we may expect the DEFAULT splitting heuristics to slow down compilation time since it generate more compilation units through splitting compared to the other three strategies. Figure 37 shows that enabling DEFAULT either leaves compilation time unchanged, or increases it by 10.7% for polymorphic benchmarks. However, enabling splitting leads to a significant 36.4% decrease of the compilation time for megamorphic benchmarks, for DEFAULT. Even more surprisingly, the impact of MISPREDICTS on the compilation time is very similar to DEFAULT: no impact overall, a 4.3% increase for minimally-polymorphic benchmarks, a larger increase of 8.6% for polymorphic benchmarks and a decrease og 37.5% for megamorphic benchmarks. Part of these results may be explained by the amount of splits performed, since MISPREDICTS fall backs to the standard splitting heuristics when a misprediction occurs, and that a high volume of mispredictions occur due to our context representation, which leads DEFAULT and MISPREDICTS to trigger more splitting than CONTEXT, therefore generating more compilation units.

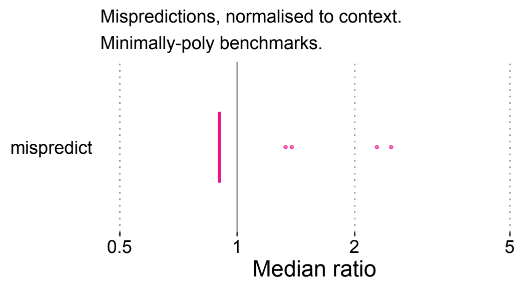
On the contrary, CONTEXT yields slow-downs of the compilation time for all benchmark categories, except polymorphic benchmarks. Since CONTEXT does not address context mispredictions, it is likely that the positive impact of reducing the amount of compilation units might be outweighed by the fact that these compilation units turn larger, or harder to optimise since they are polluted by unrelated specialisations, as the degree of polymorphism increases.

5.6 Focus on context mispredictions

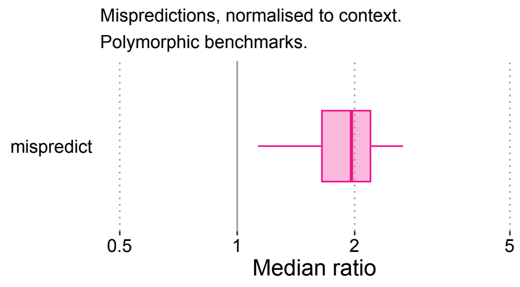
As soon as Context-Guided Splitting is enabled, either through CONTEXT or MISPREDICTS, it is possible that a context misprediction occurs, i.e., a shared method turns polymorphic because we wrongly abstracted away a context. Details of mispredictions were discussed in Section 4.2.3. In this Section, we track the amount of mispredictions and check whether it differs between CONTEXT and MISPREDICTS, since a higher amount of mispredictions may lead to longer execution times and a larger volume of splits. The results are shown in Figure 38, normalised to CONTEXT, since mispredictions do not occur in DEFAULT or NO-SPLIT.



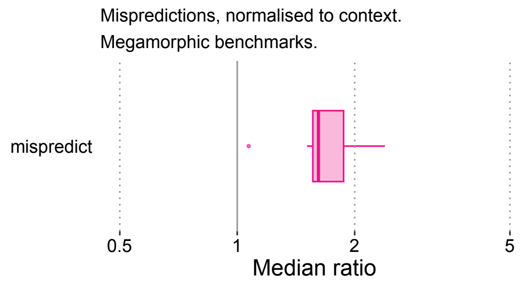
(a) All categories of benchmarks taken together. Fewer mispredictions occur in CONTEXT overall.



(b) Minimally-polymorphic benchmarks. We observe a 9.5% decrease of mispredictions for MISPREDICTS.



(c) Polymorphic benchmarks. CONTEXT is now the setting generating less mispredictions.



(d) Megamorphic benchmarks. CONTEXT is still generating less mispredictions, similarly to the polymorphic benchmarks.

Figure 38: Mispredictions, normalised to CONTEXT, on a logarithmic scale, for all categories of benchmarks. Lower is better.

Overall, MISPREDICTS triggers 12.9% (min. -9.5%, max. 2.66×) more mispredictions than CONTEXT. We observe this tendency for the polymorphic and megamorphic benchmarks (respectively, 96.9% and 61.7% more). However, CONTEXT triggers less mispredictions for the minimally-polymorphic benchmarks with a 9.5% decrease. The amount of mispredictions naturally correlates with the amount of polymorphic calls: we only use Context-Guided Splitting on methods that would be split i.e., polymorphic methods, and each call is bound to a context that might have been wrongly abstracted away. Thus, each time we dispatch to a shared method, there is a risk that a misprediction occurs and that a shared method gets invalidated.

We also measure the frequency of mispredictions—or, phrased differently, the frequency of dispatches to the "wrong" shared method—over the total amount of dispatches to a shared method, and compare the results between CONTEXT and MISPREDICTS. Their misprediction rate is very similar, at around 56.5%. The same conclusions can be drawn if we look at the benchmark category granularity: their misprediction rate stays similar for all benchmark categories. Interestingly, the minimally-polymorphic benchmarks experience a lower misprediction rate overall (20.6% for CONTEXT and 24.6% for MISPREDICTS) while the misprediction rates for the polymorphic and megamorphic benchmarks is closer to 55%. It slightly increases with the degree of polymorphism: the megamorphic benchmarks see a misprediction rate closer to 58% If we compare these results to the number of mispredictions taken in isolation, we can conclude that the number of mispredictions correlates with the number of dispatches.

Chapter 6

Discussion, Further work and Limitations

This Chapter covers the discussion, and future work, around the analysis and Context-Guided Splitting approach presented in Chapter 3, 4 and 5. Section 6.1 first discusses the trade-offs between the different splitting strategies, and explain when they are most suitable. Following with future work discussion, Section 6.2.2 then covers how the performance of Context-Guided Splitting could be improved, discussing the impact of a change of the context representation. This Section also describes alternative ways of dispatching. In the following Section 6.2.3, I present alternative ways to characterise the behaviour of programs, that are different from lookup caches. The last Section provides a comparison of our work to the existing literature.

6.1 Choosing the right splitting approach

We compared the performance of four splitting strategies in Chapter 5, focusing on each performance metric individually. The four strategies have different tradeoffs: for instance, the DEFAULT strategy performs better for peak performance, yet for the volume of AST nodes created, NO-SPLIT is the best performing strategy overall. In addition, the best performing splitting strategy per metric varies if we take the degree of polymorphism of the benchmarks into account. For instance, while DEFAULT is the best strategy regarding the time spent in garbage collection considering all benchmarks altogether, MISPREDICTS is the best performing strategy if we consider megamorphic benchmarks only, which are also the largest benchmarks in the set (see Table 1).

Several of the performance metrics we monitor are negatively correlated, which means it is unlikely a significant speedup can be observed on all the metrics at the same time. For instance, it is unlikely we can reach the same magnitude of speedup for peak and startup performance, since what is usually hampering startup performance is the profiling required to reach better performing compiled code, which dominates peak performance. Depending on the use case, it is also likely that only a subset of these performance metrics matter to the end user.

To address these different issues and offer more practical advice regarding the four splitting strategies, we propose four use-cases with different expectations regarding performance. We then suggest which splitting strategy could fit best to a given use-case when applicable. The four use-cases are described in Table 12. They are based on the three following aspects:

1. Time required to run a program, i.e., short or long-running applications;

2. Hardware constraints, e.g., cloud-hosted environments where the customer pays for more memory;

3. Codebase size, i.e., small or large programs. A small program with few dependencies is likely to display little polymorphism.

We use these metrics to build four different use-cases, labelled from `A` to `D` in the Table. The use-case `A` covers short-running applications, that are run and restarted often, typically in a pay-per-use setting, like FaaS/SaaS. These applications are

Table 12: Four use-cases representing different performance needs. We use these categories to provide recommendations regarding the different splitting strategies.

	Use-case	Performance metrics preferred	Example
A	Pay per use applications Short running applications	Startup performance Allocated memory AST size GC time	FaaS/SaaS applications
B	Long-running applications	Peak performance GC time	Web-server
C	Hardware-constrained	Allocated memory AST size	Application server Cloud-hosted applications
D	Short-running applications Small applications Monomorphic and polymorphic applications	Startup performance	Unit-testing Education

likely dominated by startup performance, and their users are likely to favour a low memory footprint to limit the cost. Use-case `B` gathers long-running applications, in a setting where the cost-related constraints are less critical than in a pay-per-use setting, e.g., using dedicated servers for instance. These applications would likely benefit from good peak performance, as well as a rather stable performance, with few pauses, that could be typically caused by GC. Use-case `C` covers applications running on a hardware-constrained environment, where costs are directly correlated to memory usage. In this setting, the performance metrics related to memory usage, i.e., the allocated memory, including AST size, are likely to matter the most. Lastly, use-case `D` represents short-running, simpler applications, where the user favours a fast development cycle, for instance, when running unit tests, or in the case of student projects. In this setting, memory usage would be less critical than in `A` or `C`, but startup performance would likely be critical.

Based on the insight of the study carried out in the previous chapters, we argue that the minimally-polymorphic benchmarks do not benefit much from any splitting strategy since they contain only a limited amount of call-sites to monomorphise. Therefore, we consider that the observed slowdowns may be caused by the extra actions as part of the splitting process, e.g., checking the lookup cache state after a call. Secondly, the megamorphic benchmarks of our set feature the largest, industry-inspired programs we studied in our experiments, but they can only represent a fraction of the behaviour of a real codebase. However, considering that megamorphism is prevalent in these benchmarks, we will assume that it is likely that real, large industry codebases contain many megamorphic call-sites too.

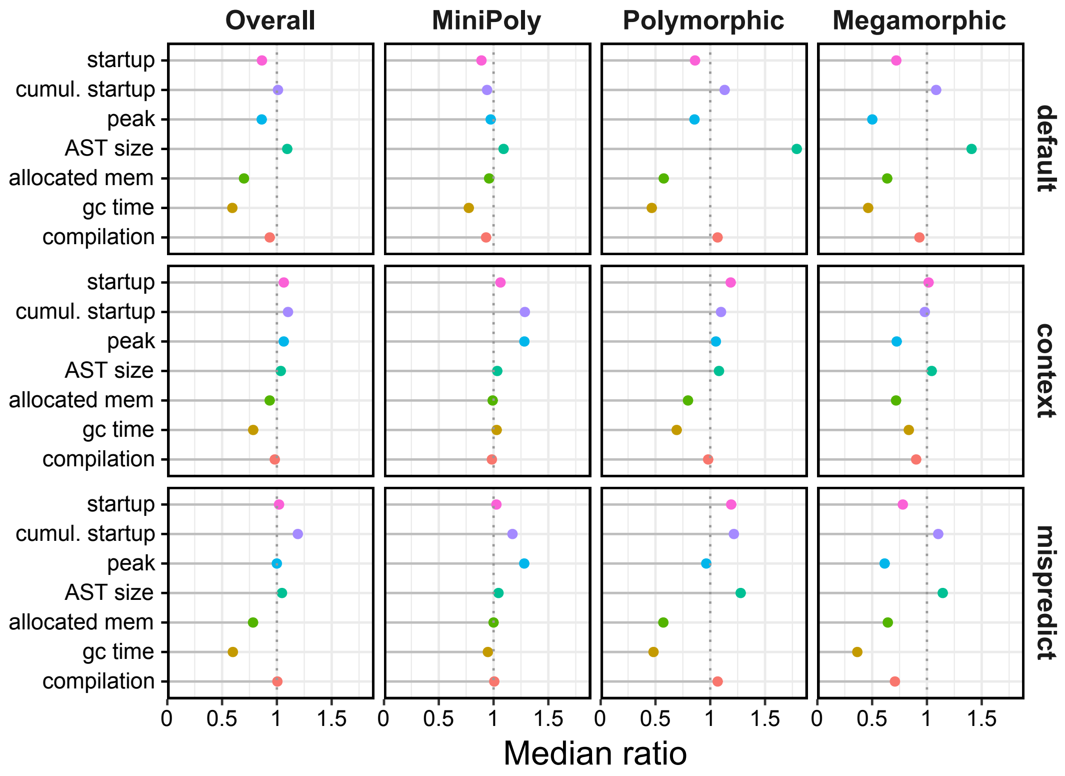


Figure 39: All performance metrics, for all categories of benchmarks. Normalised to NO-SPLIT. Lower is better.

Figure 39 provides a side-by-side comparison of the main results that have been discussed in Chapter 5, normalised to NO-SPLIT. It contains the median ratios for all splitting strategies and benchmark categories combinations. Following the four use-cases described in Figure 12, we argue that DEFAULT is likely to benefit use-case `B` the most, since it is the most impactful splitting strategy for peak performance. It also reduces the time spent in GC the most, with the exception of megamorphic benchmarks. This decrease in the time spent in GC may be beneficial in this use-case, since it might lead to a more stable execution time, with e.g., less execution time peaks due to the start of a collection, which may be desirable when a user requests a quick response time. Applications fitting in the use-case `C` would rather benefit from enabling the CONTEXT splitting strategy, which is the most impactful in terms of allocated memory, including the reduction of the size of the AST. Especially favouring startup performance, use-case `D` applications should rely on the DEFAULT splitting strategy. Regarding use-case `A`, choosing a single splitting strategy is not straightforward. Context-Guided Splitting, notably with CONTEXT, tends to benefit the most to the allocated memory and AST size metrics. Regarding startup performance, DEFAULT is the best strategy all benchmarks considered to decrease the time spent per startup iteration (see “startup” in Figure 39), but CONTEXT allows to exit the startup phase faster (see “cumul. startup” in Figure 39) for polymorphic and megamorphic benchmarks, while DEFAULT benefits for minimally-polymorphic benchmarks the most. As a consequence, for this use-case, we recommend favouring DEFAULT for the shortest-running applications, while CONTEXT should be favoured when the applications run longer.

6.2 Future work

This Section discusses some of the limitations of the experiments covered in Chapters 3 and 5, and describe possible future solutions to address these limitations.

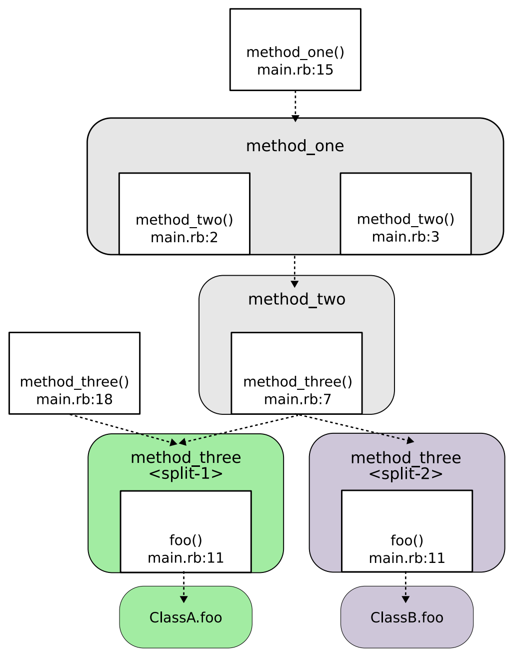
6.2.1 Experimenting with other splitting heuristics

With Context-Guided Splitting, we investigated whether the performance of splitting could be improved by adding an extra heuristic: once splitting has been decided, we use contextual insight to choose whether to split a method or to dispatch to an already split method. The default splitting strategy in TruffleRuby relies on several other assumptions, notably to decide which method to split in the first place. This decision is based on the fact that splitting is meant to eliminate polymorphism in the program, i.e., that splitting should eliminate the source of polymorphism.

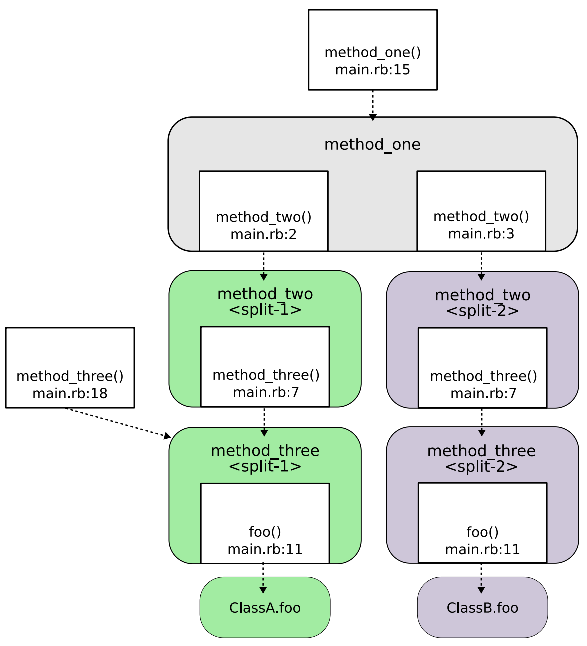
The source of polymorphism in GraalVM-based languages is identified as the location where two different call-sites point at the same method, for instance, in the case of calls to a library method from different points in the program. This assumption is used to flag a method as being a target for splitting. It is also used to propagate splitting along the call chain: the propagation stops once the source of polymorphism is reached (For further details, see Section 2.4.3 about recursive splitting). While effective at monomorphising the AST as shown in Section 3.2.2, this strategy is partly responsible for monomorphising many monomorphic methods, which we consider oversplitting.

Towards fully eliminating oversplitting by identifying alternative sources of polymorphism Figure 40 displays such a case of oversplitting. Since Context-Guided Splitting relies on the existing splitting process to flag potential splitting targets, this means that monomorphic call-sites may still be flagged as splitting targets and therefore may still be split, even though Context-Guided Splitting aims to eliminate method duplication.

The program representation on the left is an ideal one, where all redundant duplicates generated by splitting have been eliminated. Figure 40b on the right shows the same program representation resulting from applying our implementation of Context-Guided Splitting: `method_two` is monomorphic and should not have been split, but the split was necessary to eliminate the source of polymorphism, as per the rules of recursive splitting.



(a) Ideal AST structure if oversplitting as a whole is addressed.



(b) AST structure when Context-Guided Splitting is enabled. `method_two` has been split despite being monomorphic.

Figure 40: Context-Guided Splitting does not eliminate the splitting of monomorphic call-sites because it remains necessary for recursive splitting. Though, it dispatches to suitable method clones when available.

It may be worth investigating whether other strategies could help further reducing oversplitting. The assumption that polymorphism is introduced in a program by having a method called from different locations seems to stand: the results of the analysis of behaviour we carried out on Ruby programs show that all polymorphic call-sites have been eliminated after splitting was applied. Moreover, this assumption does seem to produce many false-positive, since we identified that the majority of remaining oversplitting (i.e. splitting monomorphic methods) seems to be explained by recursive splitting. We investigated whether this kind of oversplitting could be avoided by slightly modifying Context-Guided Splitting, to dispatch on specialised call-chains rather than single methods (see Section 6.2.2). While this approach is significantly lowering the amount of splitting, the preliminary results did not show a significant performance improvement.

Comparing the amount of splitting between different language implementations To get a better understanding of the use-cases leading to polymorphism and perhaps splitting, it may be worth comparing the behaviour of splitting between different language implementations. This would indicate whether specific coding practices or language constructs affect polymorphism and the splitting strategy.

This comparison needs to be carried out on a same benchmark set and equivalent splitting strategies so it is feasible to pinpoint a difference causing a change in splitting, if any. We could conduct such an experiment between different language implementations built on top of GraalVM, but would still require that we have a comparable set of benchmarks. The benchmark set used in our experiments contains the Are We Fast Yet benchmarks (Marr, Daloze and Mössenböck 2016), that have been ported in different programming languages and aim to reduce the coding differences to ease performance comparison. However, this set contains mostly micro-benchmarks, that display little polymorphism, which does not trigger splitting often. To conduct this experiment in the future, the benchmark set would need to be enriched with bigger benchmarks, that could for instance be based on the megamorphic ones of our current set, e.g., MailBench, BlogRails.

6.2.2 Improving the performance of Context-Guided Splitting

Enriching the content of a context As described in Section 4.2.1, a context is represented by a hash of the arguments' types of a call. However, this representation is limited, as shown by the volume of mispredictions generated by Context-Guided Splitting which ranges from 20.6% for minimally-polymorphic benchmarks up to 56.5% for megamorphic benchmarks (see Section 5.6), leading to reverting to the standard splitting strategy. To limit the amount of mispredictions, one could investigate whether adding further elements in a context brings any benefits. For instance, a context could also contain the current types of local and global variables during a call, the current types of fields, or the state of profiles as monitored by the system (see Section 2.3).

However, adding more criteria to the context may make the comparison more difficult if we aim at having an exact match between two contexts to allow a contextual dispatch. This in turn may lead to a even higher volume of mispredictions. Flückiger et al. (2020) address this issue by comparing contexts using a partial order, therefore dispatching to a more generic specialisation, i.e., with less assumptions, of the method if there was not a perfect match. In the future, we could define such a partial order in our system to maintain dispatching and reduce the risk to turn a shared method polymorphic. However, Flückiger et al. (2020) generate and store the split methods offline, e.g., ahead of time, in a database, which allows to store different versions of a same method, with various degrees of specialisations. Since we generate and save the split methods directly on the run-time memory, managing many different versions of a same method may prove impractical and hamper memory usage.

Investigating contexts further, identifying which aspects of an "enriched" context are more likely to influence the lookup cache contents of a method and turn it polymorphic may prove useful. It could help adapt the content of a context based on the aspects that impact significantly the degree for polymorphism.

Dispatching to call chains The current implementation of Context-Guided Splitting dispatches to a single method. In a use-case such as the one depicted in Figure 40, it means that both the `method_two` and `method_three` methods serve as dispatch locations and contain pairs of `(Context, Split methods)`. The split methods that are shared may contain calls to other methods, that may be split and/or contextually-dispatched to later on.

However, it is likely that a caller and a callee share the same specialisations.

Therefore, instead of triggering Context-Guided Splitting for each method call in the call chain, it might be beneficial to only contextually dispatch to the method at the top of a caller chain, which means that only this method would serve as a dispatch location, and would contain pairs of `(Context, Specialised caller chain)`, where a "specialised caller chain" would point at the first method of the caller chain. The dispatch location would therefore be situated right at the "source of polymorphism", which may be a single method in the case of regular splitting, or may be higher up in the caller chain if recursive splitting was triggered.

We expect this strategy to bring several benefits:

- Reduce the volume of dispatch locations required by Context-Guided Splitting;
- Further reduce the time needed to specialise methods;
- Reduce the amount of oversplitting and recursive splitting.

We conducted preliminary experiments where we dispatched to a caller chain of size two, rather than a single specialised method. This did not yield any significant performance improvements in comparison to the current implementation of Context-Guided Splitting, since it greatly increased the volume of mispredictions. Nevertheless, we believe that this experiment could lead to positive results if it was informed by additional contextual data, such as the phases described in Chapter 3.2.5. We can assume a given phase might only encounter a specific set of specialisations, and dispatch to different specialised call chains according to the execution phase the program is experiencing. Similar approaches exploiting phases, applied in different use-cases, have proven effective in terms of performance (Degenbaev et al. 2016; Gu 2007; Kistler and Franz 2003).

Duplicated methods at contextual dispatch locations Sometimes, it would be desirable that a single context dispatch to two different versions of a same method, to avoid context misprediction. Our context implementation may lead to another situation, where two different contexts dispatch to equivalent methods i.e., the two methods have a different identity, but are equivalent in terms of specialisations. For instance, `Context A` points to the method duplicate `foo <split-1>`, `Context B` points to the method duplicate `foo <split-2>`, but the lookup-caches, profiles and node specialisations of `foo <split-1>` and `foo <split-2>` are equivalent. While this should not raise an issue in terms of execution time, it may impinge on memory usage, since it would be preferable to have the two contexts pointing at the same method duplicate, and therefore eliminating the redundant second duplicate. We choose to ignore this use-case in our implementation, since the process of identifying equivalent duplicates may further stress execution time.

6.2.3 Improving the analysis of behaviour

Behaviour on larger codebases Our benchmark set is diverse and contains micro- and macro-benchmarks, some of which inspired from real industrial workload and used in Ruby implementations used in industry, e.g., TruffleRuby and the YJIT project, both used at Shopify. The largest and longest-running benchmarks of our set contain several millions calls (e.g., HexaPdf, MailBench), and often rely on large libraries, such as the Rails framework, designed for web development (e.g. BlogRails, ERubiRails). While the largest of our set, we argue these benchmarks only approximate the behaviour of a real-world application. BlogRails for instance generates a blog application receiving a few hundreds of `GET` requests, yet this only accounts for 38% of all methods contained in the code. Moreover, we experimented with the BlogRails benchmark, and built BlogRailsTwoRequests, which generates the same blog, but receives both a few hundreds of `POST` requests after the `GET` requests. We can see in Chapter 5 that the two benchmarks have a very different performance profile, notably with BlogRailsTwoRequests never reaching a steady state. Yet we argue that BlogRailsTwoRequests, while not depicting an accurate picture of a real-world workload, seems however to better represent realistic interactions that BlogRails. Moreover, it is likely the volume of requests would be several order of magnitude higher, with concurrent requests to process. Therefore, it would be desirable to test both the analysis tool we used in Chapter 3 and Context-Guided Splitting in more realistic settings.

Lookup caches as the only proxy for program behaviour We mentioned in Section 2.3 that GraalVM relies on different strategies to generate efficient specialisations. Lookup-caches are one example of these strategies, the profiling of the type of a method's arguments or the return type of a method are also other strategies used in the system. We only use the content of lookup caches in our analysis as a proxy for a program's behaviour. This has the advantage of being lightweight, and providing a good insight on the splitting behaviour too, since the default splitting process is triggered according to the cache's state. Monitoring the evolution of run-time profiling data might prove insightful: it would be interesting to check how their evolution differs from the content of lookup cache at a given call-site. This may help identify other types of patterns of behaviour, as well as explaining the high number of mispredictions Context-Guided Splitting leads to.

In addition, we identified three different behaviour patterns in our benchmark set, as described in Section 3.2.5. However, these patterns are only based on the evolution of cache content at a given call-site. Alternatively to using profile data, we could also experiment with monitoring the content of the call chain from a given call-site to determine whether other patterns of behaviour appear. Rather than monitoring the evolution of the cache content, we would therefore monitor the evolution of the chain of callers from a given call-site.

6.3 Comparison with other work

This Section describes how this work, analysing the behaviour of programs, and Context-Guided Splitting, fits with the existing literature. Section 6.3.1 revolves around the common assumptions about the behaviour of programs. In a second part (Section 6.3.2), we discuss how contextual splitting differs from existing compiler optimisations, as well as on which optimisations it is build on top of.

6.3.1 Assumptions about a program's behaviour

Characterising polymorphism Chapter 3 focuses on the call-site behaviour 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 (2015), who measure polymorphism in Python programs. Similarly to our approach, they measure the degree of receiver polymorphism (see Section 3.2.1) and target polymorphism, which they refer to as n-typeable. They also aim to guide language developers to further optimise 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 optimisations on these.

This work, as well as several others, has investigated dynamism in general, which inspires some of our methodology. Similar to the work by Åkerblom and Wrigstad (2015) on polymorphism in Python programs, we analyse the behaviour of programs written in a dynamic language using a custom, instrumented version of the runtime. We also look at similar call-site behaviours: for instance, Richards et al. (2010) investigate the validity of common assumptions about the dynamic behaviour of JavaScript programs and notably show that call-site polymorphism, which they refer to as call-site dynamism, i.e., when one call-site sees different method bodies, is more frequent than is commonly assumed (see Section 3.2.1, in Chapter 3). Holkner and Harland (2009) also investigate when and how Python programs use the more dynamic, reflection-like features, and distinguish between startup and later run time. Sarimbekov et al. (2013) analysed call-site behaviour as well, focusing on JVM languages. They include Ruby programs in their benchmark set.

Our analysis of call-site behaviour, while similar, goes beyond these works on several aspects. Notably, we study the evolution of dynamism and call-site behaviour (see Section 3.2.5), past the “startup vs. steady state” dichotomy. We also assess how some optimisations, e.g., splitting and the elimination of target duplicates, have an impact on behaviour. Lastly, our benchmark set is large (74 benchmarks in total between our two experiments), and contains several programs with millions lines of code.

Behaviour evolution at run time More recently, Sun et al. (2023) further study dynamism in large Python programs, in order to build a type system for dynamic object-oriented languages. They measure polymorphism, but analyse object construction to that regard, rather than focusing on call-sites. They also characterise how polymorphism evolves at run time from a different perspective than our study. They categorise different types of evolution (e.g., adding to, modifying types of or deleting attributes from an object) and evaluating their prevalence.

The evolution of a program’s behaviour at run time has been investigated as part of research about phases, that represent portions of the run time displaying an homogeneous behaviour. Some experiments use phases to guide dynamic optimisations, like JIT compilation (Gu 2007; Kistler and Franz 2003), or garbage collection (Wilson 1988; Ferreiro et al. 2016). We do not take advantage of phased behaviour as part of Context-Guided Splitting at the moment, but future work could investigate further whether contextual insight display a phased behaviour, and whether it could be used to better guide the trigger for splitting. As part of their research on meta-object protocols, Chari, Garbervetsky and Marr (2017) describe examples of phases based on the content of lookup caches, categorising these phases as displaying ephemeral, warm up, rare, and highly indirect variability. While they recreate these behaviours in a set of synthetic microbenchmarks, our study demonstrates such phases may be observed in much larger programs.

6.3.2 Context-Guided Splitting

Our approach regarding Context-Guided Splitting builds upon the work of Flückiger et al. (2020), later completed by Mehta et al. (2023). In their work, they propose a novel method dispatch strategy relying on contextual information. We use a more naive form of a context, relying almost exclusively on method parameters' types, while they rely on specific behavioural properties of these parameters in the context of R. In addition, they use a partial order to compare contexts while we look for an exact match. While our comparison method might less stress on execution time, it also leads to a rather high number of mispredictions that we address by reverting to a standard splitting strategy. Our approach also have different end goals: while we aim at reusing specialised methods within a single run, they ultimately aim at reusing specialised, i.e., in their case, JIT-compiled methods, across different runs. Furthermore, part of their process is performed offline, i.e., as part of a training run aiming at generated different specialisations of methods, that are then stored in a persistent database. The specialised methods we reuse are generated during the normal execution of a program, and are stored alongside the program representation.

Chambers and Ungar (1989) first described method splitting in SELF, extending it in a later publication (Chambers and Ungar 1990). In essence, splitting in TruffleRuby has the same goal of monomorphising a program, ideally enabling other optimisations such as inlining. However, TruffleRuby's splitting operates at a higher-level, on the AST, while SELF's splitting is performed on the control flow graph, the IR used by the compiler to perform many type-related optimisations.

Chapter 7

Conclusion

This thesis revolves around dynamic languages and investigates new techniques to optimise them. In these languages, the execution of method calls can be especially challenging since methods calls are late bound, i.e., the target of a call is identified at run time, during the execution of the program. Many optimisations exist to speed up the execution of method calls. Splitting, the optimisation we focus on, duplicates a given method in the program representation so that further optimisations such as inlining can be applied to improve the performance of the program. This work aims to answer the three following research questions, which focus on method calls and the ways to optimise them:

[R.Q. 1] *Do the assumptions about program behaviour used by common method call optimisations still apply in today's context?*

[R.Q. 2] *Can we trigger fewer splitting operations, all while keeping the same level of call monomorphisation and can this improve run-time metrics, such as compilation time, start-up time and memory usage?*

[R.Q. 3] *Can we re-use previous method clones resulting from splitting rather than triggering splitting again?*

To answer these questions, this work divides into two parts: first, regarding R.Q. 1, I analysed the behaviour of a large set of programs written in the dynamic programming language Ruby, which runs on top of the TruffleRuby run-time system (see Chapter 3). I tested whether the commonly accepted assumptions about a program's behaviour were still valid in today's context, since TruffleRuby is a recent runtime system, and since many of the benchmarks we use were designed and used recently as well. We tested several assumptions:

- Programs usually display low polymorphism, which means that a given call-site will usually point to the same method implementation during the whole execution.
- Existing call optimisations, such as the elimination of call target duplication and splitting, are effective in modern systems to address polymorphism.
- Method calls and closure calls behave similarly.
- The program behaviour at run time is stable.

Our analysis shows that the first two assumptions hold in today's context (see Chapter 3). However, the analysis also uncovers that albeit effective, splitting may be triggered too often, leading to potential adverse impacts in terms of execution time, memory usage and run-time compilation time.

The second part of my work therefore focuses on splitting, and how to address this apparent oversplitting, which links to R.Q. 2 and 3. To address these questions, I propose a novel splitting heuristic, called Context-Guided Splitting. I describe the principles of this technique in Chapter 4. In this approach, I use the contextual information of a method call, such as the run-time types of its arguments for instance, to determine whether a splitting operation should be triggered. If the method was previously split during the current run, a duplicate of this method is currently present in the program representation. I check whether the current context matches the one under which the split was performed, and, if it is a match, I rather dispatch to the existing duplicate than splitting again, to avoid creating a new duplicate in the program representation.

I implemented Context-Guided Splitting as part of TruffleRuby, and evaluated its performance against other splitting approaches on a large and diverse set of benchmarks. The methodology and results of this evaluation are detailed in Chapter 5. The DEFAULT splitting strategy, currently used in TruffleRuby, consistently yields good results for all benchmarks, leading to 13.9% shorter peak execution time (49.9% for megamorphic benchmarks), reducing the time spent per startup iterations within a similar range, and decreasing the volume of allocated memory by 29.9% across the whole benchmark set. We observe that Context-Guided Splitting helps to reduce oversplitting (see Section 5.2), while also frequently outshining DEFAULT when compared to a case where splitting is disabled, on various run-time metrics for our larger, more polymorphic benchmarks, which we consider to represent real-world workloads the best. Especially, it helps reducing the time spent in startup by 2% (min. -21.4%, max. 2.64×) when compared to NO-SPLIT. It also positively impacts memory usage, reducing the amount of time spent in garbage collection by 51.8% for polymorphic benchmarks, and by 56.6% for megamorphic benchmarks. Context-Guided Splitting also reduces the time spent compiling at run time, with an overall 37.5% decrease observed for megamorphic benchmarks.

The observations in our analysis of call-site behaviour were discussed with the TruffleRuby team, which led to changes on splitting for closures calls. Such calls now rely on GraalVM's default splitting heuristics rather than a custom one. We experimented with Context-Guided Splitting on TruffleRuby, but we argue that this approach could be applied and beneficial to other kinds of dynamic languages displaying similar call behaviour and features, such as, e.g., polymorphism and the use of caches. The results from Flückiger et al. (2020) show that a similar approach yields positive results in their implementation of R.

While the results of Context-Guided Splitting are promising, additional research work is needed to make Context-Guided Splitting applicable to a broader range of applications by further improving its performance. For instance, in Chapter 6, we discussed the results of our experiments on program behaviour, and provided recommendations regarding the different splitting techniques we compared. Our analysis of call-site behaviour uncovered useful insights, while solely focusing on one particular metric: the content of lookup caches. Since GraalVM-based systems like TruffleRuby rely on a wide set of profiling information to optimise a program, the lookup caches' content might not be representative of the full call-site behaviour of a program. As part of future analyses, including other proxies for call-site behaviour may help getting a finer grasp of call behaviour, which may in turn help improving Context-Guided Splitting, or uncover new optimisation opportunities. In addition, the tooling used for our call-site analysis, a instrumented version of TruffleRuby, coupled with a R application used to process and produce statistical summaries of the data, does not scale for very large execution traces. This causes an issue considering that even the largest benchmarks of our set are still moderately-sized and moderately-complex compared to a real-world workload. As a consequence, and since the analysis we conducted on call-site behaviour may also benefit from data from real-world, and large codebases, a refactoring of the tool seems necessary.

The performance of Context-Guided Splitting is promising: on some of our largest and more realistic benchmarks, guiding splitting with contextual insight helps the program to exit warm up earlier (e.g., BlogRails, LiquiCartRender, LiquidCartParse), as well as speeding up the time spent in the first run of some our benchmarks, which is critical in the context of a pay-per-use ecosystem, e.g., FaaS/SaaS. In addition, Context-Guided Splitting reduces the time spent in garbage collection for these larger benchmarks, as well as reduces the time spent compiling at run time. However, we argue that the performance of Context-Guided Splitting could be pushed further. As observed in Section 5.6, the amount of context mispredictions remains high, which suggests that solely relying on arguments' types and block identity is not sufficient to accurately describe the context of a method call. Introducing other variables in a context, such as the types of fields or globals, or other kind of profiling information may partly address that issue. Reducing the volume of mispredictions would limit the number of times Context-Guided Splitting (the MISPREDICTS approach) has to revert to the default splitting approach, and also further reduce the overall number of splits, which we expect to further improve performance, especially regarding memory usage, and possibly the time spent in the startup phase.

Furthermore, although the preliminary experiments did not yield a positive result, we argue it may still be beneficial to attempt dispatching to several already specialised methods, i.e., a caller chain for instance, rather than to a single method, like we currently do. This approach would inevitably lead to new challenges, as we would therefore need to limit mispredictions all while dealing with contexts that would apply to several different methods at once. The use of a richer contextual information may help this approach to improve performance. This might strongly relate to previous works conducted on program phases (see Section 6.3.1). One could associate a program phase to one or several contexts, and one could use this information to chose to which specialised caller-chain it is preferable to dispatch to.

Overall, modern runtimes, especially the ones targeting dynamic languages, with rather complex semantics, tend to be very effective at generating high-performance code, but are also complex pieces of software, comprised of many diverse interconnected components. GraalVM, like other meta-compilation systems, aim at limiting semantics duplication, but is still, understandably, very intricate and complex. With this work, we started to investigate the interactions between several co-existing optimisations: lookup caches, the elimination of target duplicates and splitting. Investigating how other optimisations interact with one another, as well as how such optimisations might benefit from the vast set of profiling information gathered and use on different components of the runtime system, may also be worth investigating in the future.

Bibliography

Åkerblom, B. and Wrigstad, T. (2015). Measuring polymorphism in python programs. *SIGPLAN Not*, 51(2), pp. 114–128. 144

Alpern, B. and Attanasio, C. R. (1999). Implementing Jalapeno in Java, p. 11. 34

Alpern, B. et al. (2005). The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2), pp. 399–417, conference Name: IBM Systems Journal. 28

Aumayr, D., Marr, S., Kaleba, S., Gonzalez Boix, E. and Mössenböck, H. (2021). Capturing High-level Nondeterminism in Concurrent Programs for Practical Concurrency Model Agnostic Record & Replay. *The Art, Science, and Engineering of Programming*, 5(3), p. 39.

Bala, V., Duesterwald, E. and Banerjia, S. (2000). Dynamo: A Transparent Dynamic Optimization System, p. 12. 32

Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S. and Tratt, L. (2017). Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), pp. 1–27. 98, 113

Basso, M., Bonetta, D. and Binder, W. (2023). Automatically generated supernodes for ast interpreters improve virtual-machine performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 1–13. 31

Bezanson, J., Edelman, A., Karpinski, S. and Shah, V. B. (2015). Julia: A fresh approach to numerical computing. `1411.1607`. 25

Bolz, C. F. and Tratt, L. (2015). The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98, pp. 408–421. 11, 35

Bolz, C. F., Cuni, A., Fijalkowski, M. and Rigo, A. (2009). Tracing the meta-level: PyPy's tracing JIT compiler. In I. Rogers, ed., *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'09)*, Genova, Italy: ACM Press, pp. 18–25. 32, 35

Brock, J., Ding, C., Xu, X. and Zhang, Y. (2018). PAYJIT: space-optimal JIT compilation and its practical implementation. In *Proceedings of the 27th International Conference on Compiler Construction*, Vienna, Austria: ACM Press, CC 2018, pp. 71–81. 33

Brunthaler, S. (2010). Efficient interpretation using quickening. *ACM SIGPLAN Notices*, 45(12), pp. 1–14. 31

Burchell, H., Larose, O., Kaleba, S. and Marr, S. (2023). Don't trust your profiler: An empirical study on the precision and accuracy of Java profilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, New York, NY, USA: ACM Press, MPLR 2023, pp. 100–113.

Chambers, C. and Ungar, D. (1989). Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, PLDI '89, pp. 146–160. 29, 58, 146

Chambers, C. and Ungar, D. (1990). Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *Proceedings* *of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, PLDI '90, pp. 150–164. 146

Chambers, C., Ungar, D. and Lee, E. (1989). An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10), pp. 49–70. 32, 34

Chari, G., Garbervetsky, D. and Marr, S. (2017). A metaobject protocol for optimizing application-specific run-time variability. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pp. 1–5. 145

Chevalier-Boisvert, M. (2023). Are we heading towards a dynamic language winter? (invited talk). In *Proceedings of the 19th ACM SIGPLAN International Symposium on Dynamic Languages*, New York, NY, USA: Association for Computing Machinery, DLS 2023, p. 2. 11

Chevalier-Boisvert, M. et al. (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)*, New York, NY, USA: ACM Press, pp. 25–32. 33, 46

Choi, W., Chandra, S., Necula, G. and Sen, K. (2015). Sjs: A type system for javascript with fixed object layout. In *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings 22*, Springer, pp. 181–198. 25

Click, C. and Cooper, K. D. (1995). Combining analyses, combining optimizations. *ACM Trans Program Lang Syst*, 17(2), pp. 181–196. 27

Click, C. and Paleczny, M. (1995). A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, New York, NY, USA: ACM Press, IR '95, pp. 35–49. 27

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans Program Lang Syst*, 13(4), pp. 451–490. 27

Degenbaev, U., Eisinger, J., Ernst, M., McIlroy, R. and Payer, H. (2016). Idle time garbage collection scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Santa Barbara, CA: ACM Press, pp. 570–583. 141

Deutsch, L. P. and Schiffman, A. M. (1984). Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA: ACM Press, POPL '84, pp. 297–302. 10, 28

Duboscq, G. et al. (2013). An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, New York, NY, USA: Association for Computing Machinery, VMIL '13, pp. 1–10. 27

Ertl, M. A. and Gregg, D. (2003). The Structure and Performance of Efficient Interpreters. *The Journal of Instruction-Level Parallelism*, 5, pp. 1–25. 30, 31

Evans, B. J., Gough, J. and Newland, C. (2018). *Optimizing Java: Practical Techniques for Improving JVM Application Performance*. "O'Reilly Media, Inc.". 32

Ferreiro, H., Castro, L., Janjic, V. and Hammond, K. (2016). Kindergarten cop:

dynamic nursery resizing for GHC. In *Proceedings of the 25th International Conference on Compiler Construction*, Barcelona, Spain: Association for Computing Machinery, CC 2016, pp. 56–66. 145

Flückiger, O. et al. (2020). Contextual dispatch for function specialization. *Proceedings of the ACM on Programming Languages*, 4. 15, 81, 83, 140, 146, 149

Futamura, Y. (1983). Partial computation of programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata and A. Yonezawa, eds., *RIMS Symposia on Software Science and Engineering*, Berlin, Heidelberg: Springer, Lecture Notes in Computer Science, pp. 1–35. 35

Gal, A. et al. (2009). Trace-based just-in-time type specialization for dynamic languages. *ACM SIGPLAN Notices*, 44(6), pp. 465–478. 32

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc. 44

Google (2008). Google V8 code source repository. 34

Gu, D. (2007). *Hardware related optimizations in a Java virtual machine.* Ph.D. thesis, McGill University. 141, 145

Hank, R., Hwu, W. and Rau, B. (1995). Region-based compilation: an introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 158–168, iSSN: 1072-4451. 33

Hansen, G. J. (1974). *Adaptive systems for the dynamic run-time optimization of programs.* phd, Carnegie Mellon University, USA, aAI7420364. 33, 34

Hansson, D. H. (2004). Ruby-on-Rails website. 44, 46

Hartmann, T., Noll, A. and Gross, T. (2014). Efficient code management for dynamic multi-tiered compilation systems. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pp. 51–62. 31

Holkner, A. and Harland, J. (2009). Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, AUS: Australian Computer Society, Inc., ACSC '09, pp. 19–28. 144

Hölzle, U., Chambers, C. and Ungar, D. (1991). Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, Springer, pp. 21–38. 12, 44

Ihaka, R. and Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3), pp. 299–314. 44

Ismail, M. and Suh, G. E. (2018). Quantitative overhead analysis for python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, pp. 36–47. 10

Kaleba, S., Béra, C. and Ducasse, S. (2018). Assessing primitives performance on multi-stage execution. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pp. 1–10.

Kaleba, S., Béra, C. and Miranda, E. (2018). Garbage collection evaluation infrastructure for the Cog VM. In *Proceedings of the 26th International Workshop on Smalltalk Technologies*, pp. 1–8.

Kaleba, S., Béra, C., Bergel, A. and Ducasse, S. (2017). A detailed VM profiler for the Cog VM. In *Proceedings of the 25th International Workshop on Smalltalk Technologies*, pp. 1–8.

Kaleba, S., Larose, O., Jones, R. and Marr, S. (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*, Auckland, New Zealand: ACM Press, pp. 15–28.

Kistler, T. and Franz, M. (2003). Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4), pp. 500–548. 141, 145

Knuth, D. E. (1971). An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2), pp. 105–133, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380010203. 34

Kulkarni, P. A. (2011). JIT compilation policy for modern machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, Portland, Oregon, USA: ACM Press, OOPSLA '11, pp. 773–788. 33

Larose, O., Kaleba, S. and Marr, S. (2022). Less is more: Merging AST Nodes to Optimize Interpreters. 31

Larose, O., Kaleba, S., Burchell, H. and Marr, S. (2023). AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. New York, NY, USA: ACM Press. 126

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, USA: IEEE Computer Society, CGO '04, p. 75. 28

Lee, S.-W., Moon, S.-M. and Kim, S.-M. (2008). Enhanced Hot Spot Detection Heuristics for Embedded Java Just-in-Time Compilers, p. 10. 34

Lewis, B., LaLiberte, D. and Stallman, R. (1990). The GNU Manual Group. *GNU Emacs Lisp Reference Manual, for Emacs Version*, 21, pp. 02111–1307. 10, 25

Marr, S. (2023). ReBench: Execute and Document Benchmarks Reproducibly. Version 1.2. 98

Marr, S., Daloze, B. and Mössenböck, H. (2016). Cross-language compiler benchmarking: are we fast yet? *ACM SIGPLAN Notices*, 52(2), pp. 120–131. 45, 139

Matsumoto, Y. and Ishituka, K. (2002). Ruby programming language. 44

Mehta, M. K., Krynski, S., Gualandi, H. M., Thakur, M. and Vitek, J. (2023). Reusing just-in-time compiled code. New York, NY, USA: ACM Press. 146

Menard, K. (2023). JITs are nice, but why aren't we using them? (invited talk). In *Proceedings of the 19th ACM SIGPLAN International Symposium on Dynamic Languages*, New York, NY, USA: Association for Computing Machinery, DLS 2023, p. 4. 11

Milton, S., Schmidt, H. et al. (1994). Dynamic dispatch in object-oriented languages. 25

Moon, D. A. (1974). *MACLISP reference manual.* Massachusetts Institute of Technology. 10, 25

Nagpurkar, P. (2007). *Analysis, Detection, and Exploitation of Phase Behavior in Java Programs.* Ph.D. thesis, University of California Santa Barbara. 44, 65

Ottoni, G. (2018). HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*, Philadelphia, PA, USA: ACM Press, pp. 151–165. 33

Paleczny, M., Vick, C. and Click, C. (2001). The Java HotSpotTM Server Compiler, p. 13. 34

Perelman, E. et al. (2006). Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, Rhodes Island, Greece: IEEE, p. 10 pp. 44, 65

Piumarta, I. and Riccardi, F. (1998). Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 291–300. 31

Pontelli, E., Ranjan, D. and Gupta, G. (1998). The complexity of late-binding in dynamic object-oriented languages. In *International Conference on Algebraic and Logic Programming*, Springer, pp. 213–229. 25

Richards, G., Lebresne, S., Burg, B. and Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM Press, PLDI '10, pp. 1–12. 144

Romer, T. H. et al. (1996). The structure and performance of interpreters. *ACM SIGPLAN Notices*, 31(9), pp. 150–159. 30

Sarimbekov, A. et al. (2013). Characteristics of dynamic jvm languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, New York, NY, USA: Association for Computing Machinery, VMIL '13, pp. 11–20. 53, 145

Scheifler, R. W. (1977). An analysis of inline substitution for a structured programming language. *Commun ACM*, 20(9), pp. 647–654. 10, 30

Schilling, J. L. (2003). The simplest heuristics may be the best in Java JIT compilers. *ACM SIGPLAN Notices*, 38(2), pp. 36–46. 31, 34

Serrano, M. (2018). JavaScript AOT Compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, Boston, MA: ACM Press, pp. 50–63. 25

Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H. and Nakatani, T. (2005). Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4), pp. 732–785. 34

Sun, K., Chen, S., Wang, M. and Hao, D. (2023). What types are needed for typing dynamic objects? a python-based empirical study. In C.-K. Hur, ed., *Programming Languages and Systems*, Singapore: Springer Nature Singapore, pp. 24–45. 145

Tadeu Zagallo (2019). A New Bytecode Format for JavaScriptCore. 31

Ungar, D. and Smith, R. B. (1987). Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 227–242. 10, 25

Whaley, J. (2000). A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*, San Francisco, California, United States: ACM Press, pp. 78–87. 34

Wilson, P. R. (1988). Opportunistic garbage collection. *ACM SIGPLAN Notices*, 23(12), pp. 98–102. 145

Wimmer, C. and Würthinger, T. (2012). Truffle: a self-optimizing runtime system.

In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, Tucson, Arizona, USA: ACM Press, SPLASH '12, pp. 13–14. 35

Wimmer, C. et al. (2019). Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), pp. 1–29. 25

Würthinger, T. et al. (2012). Self-optimizing AST interpreters. In *8th ACM SIGPLAN International Symposium on on Dynamic Languages*, Tucson, AZ: ACM Press, pp. 73–82. 12, 31, 35, 86

Würthinger, T. et al. (2013). One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! '13*, Indianapolis, Indiana, USA: ACM Press, pp. 187–204. 11

Würthinger, T. et al. (2017). Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, Barcelona, Spain: ACM Press, pp. 662–676. 36

Yvon, S. and Feeley, M. (2021). A small Scheme VM, compiler, and repl in 4k. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pp. 14–24. 25

Zendra, O. and Driesen, K. (2002). Stress-testing control structures for dynamic dispatch in java. In *2nd Java Virtual Machine Research and Technology Symposium (Java VM 02).* 25

Zheng, Y., Bulej, L. and Binder, W. (2017). An Empirical Study on Deoptimization in the Graal Compiler. In P. Müller, ed., *31st European Conference on*

Object-Oriented Programming, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 74, Barcelona, Spain: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 30:1–30:30. 31

Appendix A

Do the common assumptions about run-time behaviour still hold? - additional material

A.1 Description of the benchmark set

Table 13: Description of our benchmark set

Benchmark	**Description**
Acid	Stresses the meta-programming capabilities of Ruby
AsciiDoctorLoadFile	Parses a text file into an an annotated tree
AsciiDoctorConvert	Traverses an annotated tree and IO writing
BinaryTrees	Creates binary trees and discards them
BlogRails	Simulates a blog application (access to the posts)
BlogRailsTwoRequests	Simulates a blog application (access and update to the posts))
Bounce	Simulates a box of bouncing balls
CD	Simulates an airplane collision detector
ChunkyPNG*	Stresses the ChunkyPNG Ruby library (Read, write and manipulate PNG files)
DeltaBlue	Implements a constraint-solver
ErubiRails	Renders a Discourse view (String concatenation)
FannkuchRedux	Performs multiple accesses and swaps between elements of small arrays
Havlak	Implements a loop recognition algorithm
HexaPDF(Small)	Wraps a txt file into a pdf (variant with a smaller text)
ImageDemo*	Processes an image using the Sobel operator
JSON	Parses JSON strings
Lee	Lays a circuit board using the Lee algorithm
LiquidCartParse	Parses a Liquid HTML template for a shopping cart
LiquidCartRender	Renders a shopping cart using Liquid HTML templates
LiquidMiddleware	Parses and renders an simple inlined Liquid template
LiquidParseAll	Parses many Liquid HTML templates
LiquidRenderBibs	Renders a bibliography using Liquid HTML templates
List	Creates and traverses lists
MailBench	Parses emails
Mandelbrot	Generates fractals
MatrixMultiply	Multiplies two matrices together
NBody	Models the orbits of planets around the sun
NeuralNet	Simulates a neural network
OptCarrot	Emulates a NES
Permute	Generates permutations of an array
PiDigits	Generates digits of Pi
PSD*	Stresses the PSD ruby library (Read, write and manipulate PSD files)
PsychLoad	Parses a YAML file
Queens	Creates a solver for the Eight Queens problem
RedBlack	Creates and traverses a red-black tree
Richards	Simulates an OS kernel
Rubykon	Creates an AI from the go board game
Sieve	Implements the sieve of Eratosthenes
Sinatra	Creates and interacts with a web application. Uses the Ruby Sinatra framework
SpectralNorm	Calculates the spectral norm of a matrix
Storage	Creates and deletes trees of arrays
Towers	Covers the tower of Hanoi problem

A.2 Impact of Call-Site Optimisations on Polymorphic and Megamorphic Calls

Table 14: Full table displaying the impact of eliminating target duplicates on polymorphism: CD does not see target duplicates, but the other minimally-polymorphic benchmarks see at least 50% of their polymorphic calls monomorphised.

	Number of calls		**Change 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 15: Full table displaying the impact of splitting on polymorphism: it fully addresses the remaining polymorphism in minimally-polymorphic benchmarks.

	Number of calls		**Change 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

A.3 Impact of Call-Site Optimisations on Polymorphic and Megamorphic Call-Sites

Table 16: Full table displaying the impact of eliminating target duplicates in the cache on the amount of polymorphic and 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 17: 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		**Change 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

Appendix B

Results - additional material

B.1 Benchmark configuration file

Listing B.1: ReBench configuration file needed to run the performance experiments from Chapter 5. The number of iterations per benchmarks are displayed in lines 18–84. The execution flags for each run are available in lines 87–106.

```
1  # -*- mode: yaml -*-
2  # Config file for ReBench
3  default_experiment: benchmarks
4  default_data_file: 'splitting-benchmark.data'
5  
6  # definition of benchmark suites
7  benchmark_suites:
8      bundle-based-apps:
9          gauge_adapter: RebenchLog
10         location: .
11         command: &HARNESS_CMD "%(variable)s ./phd-bench/phase/harness-full.rb %(
               benchmark)s %(iterations)s"
12         iterations: 1
13         invocations: 50
14         max_invocation_time: 3024000
```

```
        benchmarks:
            - LeeBench: {extra_args: 300, machines: [yuria2 ]}

    ruby-benchs-steady:
        gauge_adapter: RebenchLog
        location: .
        command: *HARNESS_CMD
        iterations: 1
        invocations: 100
        max_invocation_time: 3024000
        benchmarks:
            - Acid: {extra_args: 5000, machines: [yuria2 ]}
            - AsciidoctorConvertSmall: {extra_args: 100, machines: [yuria3 ]}
            - AsciidoctorLoadFileSmall: {extra_args: 100, machines: [yuria ]}
            - BinaryTrees: {extra_args: 1, machines: [yuria2 ]}
            - ImageDemoConv: {extra_args: 30, machines: [yuria3 ]}
            - ImageDemoSobel: {extra_args: 1, machines: [yuria ]}
            - LiquidCartParse: {extra_args: 300, machines: [yuria2 ]}
            - LiquidCartRender: {extra_args: 900, machines: [yuria3 ]}
            - LiquidMiddleware: {extra_args: 12000, machines: [yuria ]}
            - LiquidParseAll: {extra_args: 900, machines: [yuria2 ]}
            - LiquidRenderBibs: {extra_args: 900, machines: [yuria3 ]}
            - MatrixMultiply: {extra_args: 200, machines: [yuria2 ]}
            - OptCarrot: {extra_args: 100, machines: [yuria3 ]}
            - Pidigits: {extra_args: 2, machines: [yuria ]}
            - RedBlack: {extra_args: 5, machines: [yuria2 ]}
            - SinatraHello: {extra_args: 2000, machines: [yuria3 ]}


    game-benchs-steady:
        gauge_adapter: RebenchLog
        location: .
        command: *HARNESS_CMD
        iterations: 1
    invocations: 100
    max_invocation_time: 3024000
    benchmarks:
        - Bounce: {extra_args: 2000, machines: [yuria ]}
        - CD: {extra_args: 100, machines: [yuria2 ]}
        - DeltaBlue: {extra_args: 9000, machines: [yuria3 ]}
        - Json: {extra_args: 120, machines: [yuria ]}
        - List: {extra_args: 500, machines: [yuria3 ]}
        - Mandelbrot: {extra_args: 500, machines: [yuria2 ]}
        - NBody: {extra_args: 250000, machines: [yuria ]}
        - NeuralNet: {extra_args: 1, machines: [yuria3 ]}
        - Permute: {extra_args: 500, machines: [yuria2 ]}
        - Queens: {extra_args: 300, machines: [yuria ]}
        - Richards: {extra_args: 15, machines: [yuria2 ]}
        - Sieve: {extra_args: 5000, machines: [yuria3 ]}
        - SpectralNorm: {extra_args: 1, machines: [yuria2 ]}
        - Storage: {extra_args: 300, machines: [yuria3 ]}
        - Towers: {extra_args: 200, machines: [yuria ]}

startup-exp:
    gauge_adapter: RebenchLog
    location: .
    command: *HARNESS_CMD
    iterations: 1
    invocations: 50
    max_invocation_time: 3024000
    benchmarks:
        - FannkuchRedux: {extra_args: 1, machines: [yuria ]}
        - RubykonBench: {extra_args: 1, machines: [yuria2 ]}
        - BlogRailsRoutes: {extra_args: 1, machines: [yuria ]}
        - BlogRailsRoutesTwoRoutesTwoRequests: {extra_args: 1, machines: [yuria ]}
        - ERubiRails: {extra_args: 1, machines: [yuria2 ]}
        - MailBench: {extra_args: 1, machines: [yuria3 ]}
        - PsychLoad: {extra_args: 1, machines: [yuria ]}
```

```
      - HexaPdfSmall: {machines: [yuria2 ], extra_args: 1}
      - Havlak: {extra_args: 1, machines: [yuria3], variable_values: ["--vm.Xss5m"]}

executors:
  TruffleRuby-ce:
    path: /tmp/truffleruby/truffleruby-jvm-ce/bin
    executable: truffleruby
    args: "--experimental-options --vm.Dpolyglot.engine.TraceSplittingSummary=false
      --vm.Dpolyglot.engine.CompilationStatistics=false --vm.Dpolyglot.engine.
      DynamicCompilationThresholds=false --vm.XX:-UseJVMCINativeLibrary --
      engine.BackgroundCompilation=false --vm.Dpolyglot.log.file='tmplog'"

  TruffleRuby-ce-large-cache:
    path: /tmp/truffleruby/truffleruby-jvm-ce/bin
    executable: truffleruby
    args: "--experimental-options --vm.XX:ReservedCodeCacheSize=512m --vm.XX:
      NonProfiledCodeHeapSize=500m --vm.Dpolyglot.engine.TraceSplittingSummary=
      false --vm.Dpolyglot.engine.CompilationStatistics=false --vm.Dpolyglot.engine.
      DynamicCompilationThresholds=false --vm.XX:-UseJVMCINativeLibrary --
      engine.BackgroundCompilation=false --vm.Dpolyglot.log.file='tmplog'"

  No-Splitting-TruffleRuby-ce:
    path: /tmp/truffleruby/truffleruby-jvm-ce/bin
    executable: truffleruby
    args: "--experimental-options --vm.Dpolyglot.engine.TraceSplittingSummary=false
      --vm.Dpolyglot.engine.CompilationStatistics=false --vm.Dpolyglot.engine.
      DynamicCompilationThresholds=false --vm.XX:-UseJVMCINativeLibrary --
      engine.BackgroundCompilation=false --vm.Dpolyglot.engine.Splitting=false --vm.
      Dpolyglot.log.file='tmplog'"

  No-Splitting-TruffleRuby-ce-large-cache:
    path: /tmp/truffleruby/truffleruby-jvm-ce/bin
    executable: truffleruby
    args: "−−experimental−options −−vm.XX:ReservedCodeCacheSize=512m −−vm.XX:
        NonProfiledCodeHeapSize=500m −−vm.Dpolyglot.engine.TraceSplittingSummary=
        false −−vm.Dpolyglot.engine.CompilationStatistics=false −−vm.Dpolyglot.engine.
        DynamicCompilationThresholds=false −−vm.XX:−UseJVMCINativeLibrary −−
        engine.BackgroundCompilation=false −−vm.Dpolyglot.engine.Splitting=false −−vm.
        Dpolyglot.log.file='tmplog'"

experiments:
  benchmarks:
    description: All benchmarks
    executions:
      − TruffleRuby−ce:
          suites:
            − ruby−benchs−steady
            − game−benchs−steady
            − bundle−based−apps

      − TruffleRuby−ce−large−cache:
          suites:
            − startup−exp

      − No−Splitting−TruffleRuby−ce:
          suites:
            − ruby−benchs−steady
            − game−benchs−steady
            − bundle−based−apps

      − No−Splitting−TruffleRuby−ce−large−cache:
          suites:
            − startup−exp
```

B.2 Impact of Context-Guided Splitting on behaviour

B.2.1 Lookup caches' state

Table 18: Full table showing the impact of the CONTEXT splitting strategy on the monomorphisation of minimally-polymorphic, polymorphic and megamorphic calls. All remaining polymorphism is eliminated by CONTEXT in the minimally-polymorphic benchmarks.

	Number of calls		Change after splitting		Number
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**	**of splits**
BlogRails	799,623	190	-98.5%	-100%	1198
ChunkyCanvas*	228	0	-100%	0%	30
ChunkyColor*	228	0	-100%	0%	30
ChunkyDec	228	0	-100%	0%	30
ERubiRails	211,799	160	-99.8%	-100%	950
HexaPdfSmall	1,713,811	2,066	-100%	-100%	605
Lee	12,184	0	-100%	0%	381
LiquidCartParse	677	0	-100%	0%	71
LiquidCartRender	2,534	0	-100%	0%	133
LiquidMiddleware	650	0	-100%	0%	76
LiquidParseAll	1,079	0	-100%	0%	78
LiquidRenderBibs	32,807	0	-100%	0%	113
MailBench	36,764	0	-100%	0%	490
PsdColor	7,724	0	-100%	0%	202
PsdCompose*	7,724	0	-100%	0%	202
PsdImage*	7,730	0	-100%	0%	202
PsdUtil*	7,720	0	-100%	0%	202
Sinatra	2,274	220	-94.6%	-100%	170
ADConvert	12,127	0	-100%	0%	128
ADLoadFile	10,104	0	-100%	0%	105
DeltaBlue	634	0	-100%	0%	52
OptCarrot	18,167	0	-100%	0%	55
RedBlack	8,015,482	0	-100%	0%	40
Acid	98	0	-100%	0%	20
BinaryTrees	98	0	-100%	0%	20
Bounce	102	0	-100%	0%	20
Fannkuch	98	0	-100%	0%	20
Havlak	667,006	0	-100%	0%	32
ImgDemoConv	102	0	-100%	0%	20
ImgDemoSobel	106	0	-100%	0%	20
Json	102	0	-100%	0%	20
List	98	0	-100%	0%	20
Mandelbrot	98	0	-100%	0%	20
MatrixMultiply	98	0	-100%	0%	20
NBody	98	0	-100%	0%	20
NeuralNet	140	0	-100%	0%	36
Permute	98	0	-100%	0%	20
Pidigits	98	0	-100%	0%	20
Queens	98	0	-100%	0%	20
Richards	98	0	-100%	0%	21
Sieve	98	0	-100%	0%	20
SpectralNorm	98	0	-100%	0%	25
Storage	102	0	-100%	0%	20
Towers	98	0	-100%	0%	20

Table 19: Full table showing the impact of the MISPREDICTS splitting strategy on the monomorphisation of minimally-polymorphic, polymorphic and megamorphic calls. All remaining megamorphism is eliminated by MISPREDICTS. Like CONTEXT, a marginal number of polymorphic calls remain in BlogRailsRoutes, ERubiRails and Sinatra, our megamorphic benchmarks. All polymorphism is eliminated from minimally-polymorphic benchmarks. The number of splits is higher than CONTEXT since a context misprediction will revert to the default splitting strategy.

	Number of calls		**Change after splitting**		**Number**
Benchmark	**Poly.**	**Mega.**	**Poly.**	**Mega.**	**of splits**
BlogRails	801,093	190	-98.7%	-100%	3294
ChunkyCanvas*	228	0	-100%	0%	38
ChunkyColor*	228	0	-100%	0%	38
ChunkyDec	228	0	-100%	0%	38
ERubiRails	211,372	160	-99.8%	-100%	2214
HexaPdfSmall	1,713,777	2,066	-100%	-100%	1134
Lee	12,110	0	-100%	0%	658
LiquidCartParse	677	0	-100%	0%	128
LiquidCartRender	2,534	0	-100%	0%	260
LiquidMiddleware	650	0	-100%	0%	111
LiquidParseAll	1,079	0	-100%	0%	162
LiquidRenderBibs	31,492	0	-100%	0%	205
MailBench	36,210	0	-100%	0%	909
PsychLoad	116,055	0	-100%	0%	694
Sinatra	2,163	220	-99.5%	-100%	345
ADConvert	12,127	0	-100%	0%	449
ADLoadFile	10,104	0	-100%	0%	363
DeltaBlue	634	0	-100%	0%	74
OptCarrot	6,452	0	-100%	0%	101
RedBlack	8,167,013	0	-100%	0%	47
Acid	98	0	-100%	0%	27
BinaryTrees	98	0	-100%	0%	27
Bounce	102	0	-100%	0%	27
Fannkuch	98	0	-100%	0%	27
Havlak	667,006	0	-100%	0%	46
ImgDemoConv	102	0	-100%	0%	27
ImgDemoSobel	106	0	-100%	0%	27
Json	102	0	-100%	0%	27
List	98	0	-100%	0%	27
Mandelbrot	98	0	-100%	0%	27
MatrixMultiply	98	0	-100%	0%	27
NBody	98	0	-100%	0%	27
NeuralNet	140	0	-100%	0%	43
Permute	98	0	-100%	0%	27
Pidigits	98	0	-100%	0%	27
Queens	98	0	-100%	0%	27
Richards	98	0	-100%	0%	28
Sieve	98	0	-100%	0%	27
SpectralNorm	98	0	-100%	0%	32
Storage	102	0	-100%	0%	27
Towers	98	0	-100%	0%	27

B.2.2 Focus on splitting transitions

Table 20: Full table showing the splitting transitions for the CONTEXT strategy, for all categories of benchmarks. The minimally-polymorphic benchmarks behave homogeneously, all seeing around 90% of their splits leading to lookup caches containing the same targets.

		Cache entries after splitting (% of total number of splits)	
Benchmark	**Number of splits**	**Different**	**Same**
BlogRails	1198	24%	76%
ChunkyCanvas*	30	10%	90%
ChunkyColor*	30	10%	90%
ChunkyDec	30	10%	90%
ERubiRails	950	23%	77%
HexaPdfSmall	605	20%	80%
Lee	381	19%	81%
LiquidCartParse	71	8%	92%
LiquidCartRender	133	11%	89%
LiquidMiddleware	76	13%	87%
LiquidParseAll	78	9%	91%
LiquidRenderBibs	113	17%	83%
MailBench	490	18%	82%
PsdColor	202	17%	83%
PsdCompose*	202	17%	83%
PsdImage*	202	17%	83%
PsdUtil*	202	17%	83%
Sinatra	170	19%	81%
ADConvert	128	25%	75%
ADLoadFile	105	22%	78%
DeltaBlue	52	42%	58%
OptCarrot	55	16%	84%
RedBlack	40	40%	60%
Acid	20	10%	90%
BinaryTrees	20	10%	90%
Bounce	20	10%	90%
Fannkuch	20	10%	90%
Havlak	32	9%	91%
ImgDemoConv	20	10%	90%
ImgDemoSobel	20	10%	90%
Json	20	10%	90%
List	20	10%	90%
Mandelbrot	20	10%	90%
MatrixMultiply	20	10%	90%
NBody	20	10%	90%
NeuralNet	36	8%	92%
Permute	20	10%	90%
Pidigits	20	10%	90%
Queens	20	10%	90%
Richards	21	10%	90%
Sieve	20	10%	90%
SpectralNorm	25	8%	92%
Storage	20	10%	90%
Towers	20	10%	90%

Table 21: Full table showing the splitting transitions for the MISPREDICTS strategy, for all categories of benchmarks. Similarly to CONTEXT, the minimally-polymorphic benchmarks behave homogeneously, and see a slightly higher proportion of splitting leading to unchanged cache content, at around 93%.

		Cache entries after splitting (% of total number of splits)	
Benchmark	**Number of splits**	**Different**	**Same**
BlogRails	3294	9%	91%
ChunkyCanvas*	38	8%	92%
ChunkyColor*	38	8%	92%
ChunkyDec	38	8%	92%
ERubiRails	2214	10%	90%
HexaPdfSmall	1134	11%	89%
Lee	658	11%	89%
LiquidCartParse	128	5%	95%
LiquidCartRender	260	6%	94%
LiquidMiddleware	111	9%	91%
LiquidParseAll	162	4%	96%
LiquidRenderBibs	205	9%	91%
MailBench	909	10%	90%
PsychLoad	694	11%	89%
Sinatra	345	9%	90%
ADConvert	449	7%	93%
ADLoadFile	363	6%	94%
DeltaBlue	74	30%	70%
OptCarrot	101	9%	91%
RedBlack	47	34%	66%
Acid	27	7%	93%
BinaryTrees	27	7%	93%
Bounce	27	7%	93%
Fannkuch	27	7%	93%
Havlak	46	7%	93%
ImgDemoConv	27	7%	93%
ImgDemoSobel	27	7%	93%
Json	27	7%	93%
List	27	7%	93%
Mandelbrot	27	7%	93%
MatrixMultiply	27	7%	93%
NBody	27	7%	93%
NeuralNet	43	7%	93%
Permute	27	7%	93%
Pidigits	27	7%	93%
Queens	27	7%	93%
Richards	28	7%	93%
Sieve	27	7%	93%
SpectralNorm	32	6%	94%
Storage	27	7%	93%
Towers	27	7%	93%

B.3 Detailed plots regarding startup performance

B.3.1 Duration of the startup phase

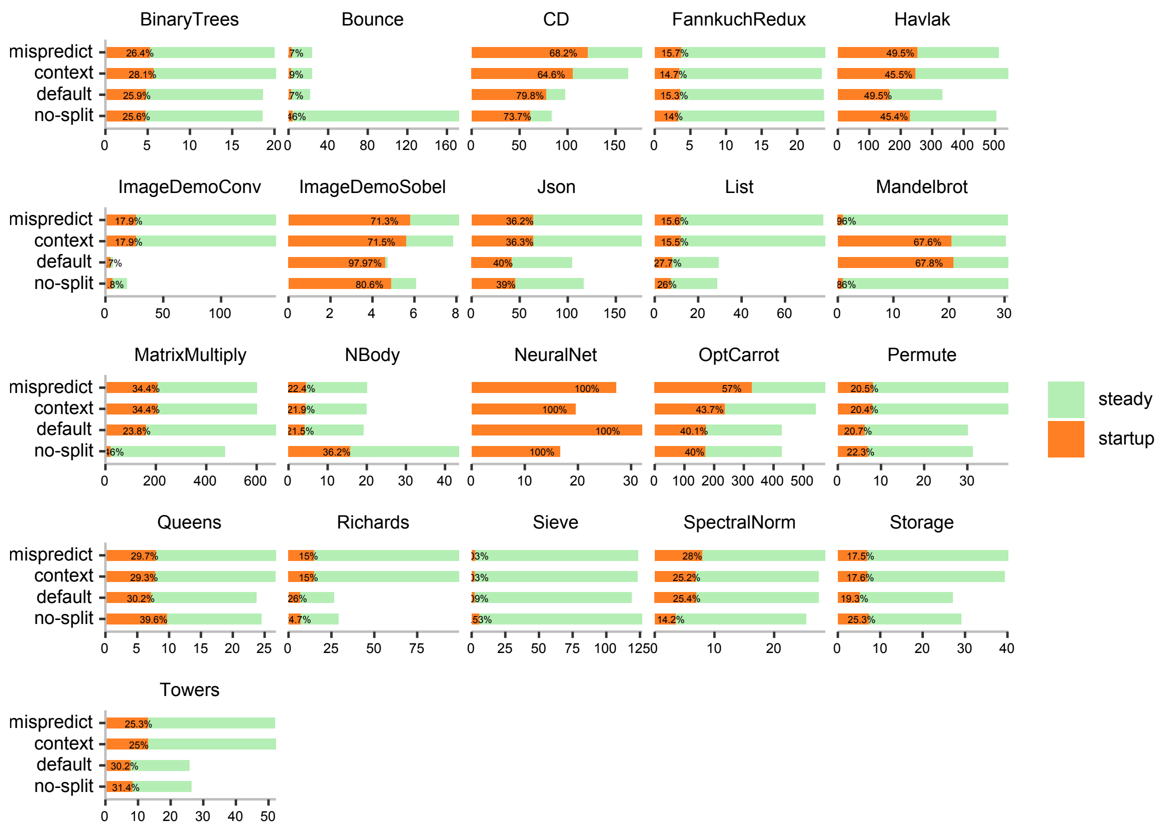


Figure 41: The four splitting strategies have varying impact on the duration of the startup phase in our benchmarks. Minimally-polymorphic benchmarks only.

B.3.2 Focus on the first iteration

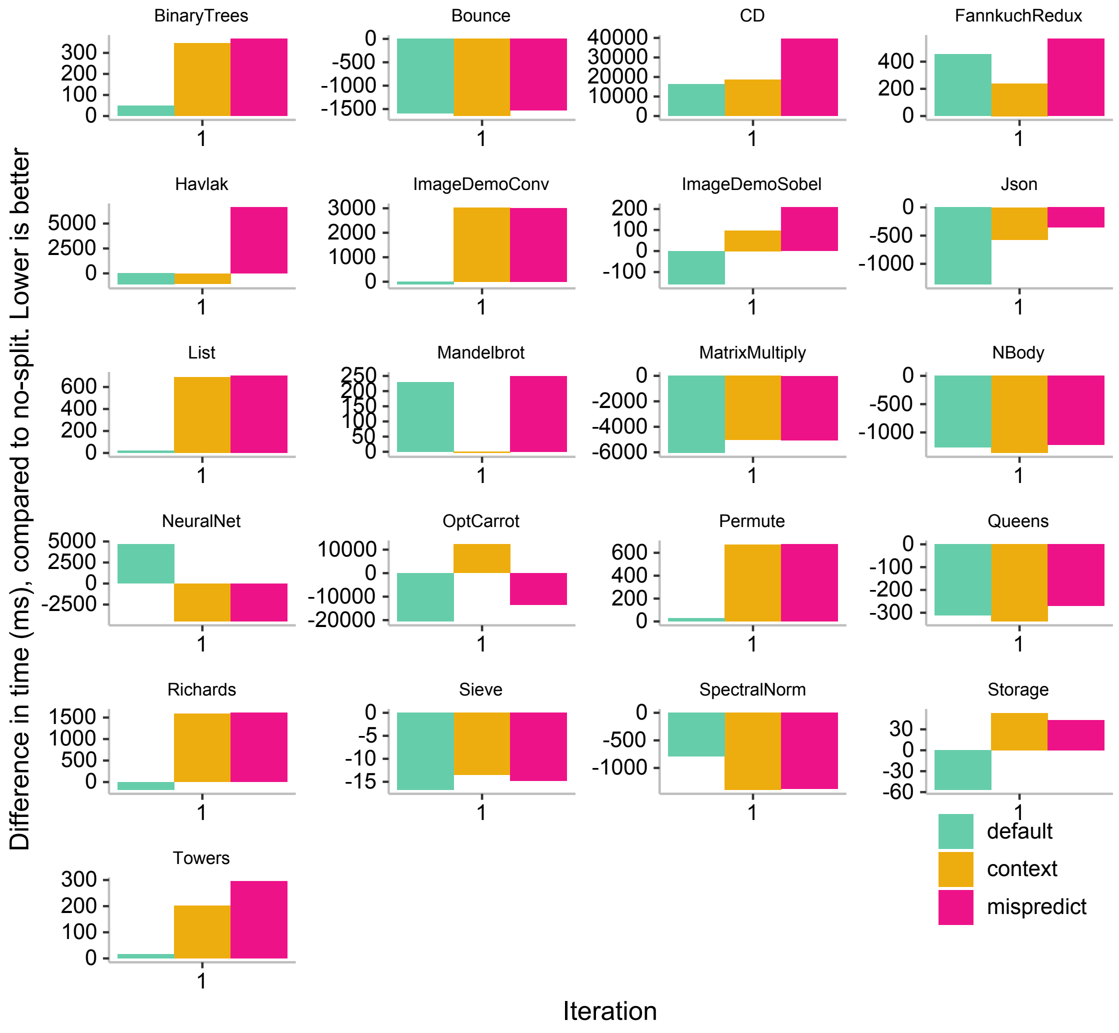


Figure 42: Impact of the splitting strategies on the time spent in the first iteration of the benchmarks. Minimally-polymorphic benchmarks only.

B.4 Detailed tables regarding the performance of the four splitting strategies

B.4.1 Execution time, startup excluded

Table 22: Median execution time per iteration, startup excluded. All benchmarks, normalised to osplit.

Benchmark	VM	median	min	max	ratio
ADConvert	no-split	3516.19	3513.26	5098.9	1
ADConvert	default	3029.2	3027.21	3165.2	0.861
ADConvert	context	3555.53	3552.47	5160.7	1.011
ADConvert	mispredict	3350.62	3348.37	3485.3	0.953
ADLoadFile	no-split	1829.97	1815.21	3392.9	1
ADLoadFile	default	1566.76	1560.89	1705.3	0.856
ADLoadFile	context	1923.73	1915.72	3467	1.051
ADLoadFile	mispredict	1763.48	1757.83	1900.2	0.964
BinaryTrees	no-split	97.39	93.69	99.8	1
BinaryTrees	default	97.38	95.92	99.9	1
BinaryTrees	context	103.5	101.49	105.3	1.063
BinaryTrees	mispredict	103.86	99.67	105.4	1.066
BlogRails	no-split	6835.62	6782.47	7037.9	1
BlogRails	default	3360.25	3319.73	7181.7	0.492
BlogRails	context	4910.76	4817.66	8588.7	0.718
BlogRails	mispredict	4039.67	3970.29	7653.3	0.591
Bounce	no-split	563.87	561.86	657.5	1
Bounce	default	62.91	62.02	153.9	0.112
Bounce	context	69.54	68.81	163.8	0.123
Bounce	mispredict	69.54	68.82	163.9	0.123
CD	no-split	109.18	108.02	194.6	1
CD	default	98.43	97.41	183	0.902
CD	context	290.28	288.6	380.1	2.659
CD	mispredict	283.67	283.33	370.8	2.598

Table 22: Median execution time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
DeltaBlue	no-split	170.79	160.94	1528.1	1
DeltaBlue	default	74.56	69.26	450.8	0.437
DeltaBlue	context	142.41	135.26	1636.8	0.834
DeltaBlue	mispredict	116.76	111.44	2488.2	0.684
FannkuchRedux	no-split	425.36	418.65	434.1	1
FannkuchRedux	default	418.75	413.4	428.4	0.984
FannkuchRedux	context	416.02	412.29	423.8	0.978
FannkuchRedux	mispredict	421.11	414.41	430.5	0.99
Havlak	no-split	915.49	911.1	1325.5	1
Havlak	default	554.98	550.94	1055.5	0.606
Havlak	context	980.09	972.91	1499.4	1.071
Havlak	mispredict	874.46	869.9	1305.3	0.955
HexaPdfSmall	no-split	1498.72	1474.78	3964.6	1
HexaPdfSmall	default	922.16	909.59	2074.6	0.615
HexaPdfSmall	context	1380.72	1351.87	5144.1	0.921
HexaPdfSmall	mispredict	1226.37	1201.03	6609.4	0.818
ImageDemoConv	no-split	94.24	93.53	95.8	1
ImageDemoConv	default	14.28	14.27	14.9	0.152
ImageDemoConv	context	938.22	935.1	943.4	9.956
ImageDemoConv	mispredict	938.65	935.95	945.3	9.961
ImageDemoSobel	no-split	7.87	7.69	10.8	1
ImageDemoSobel	default	0.64	0.63	0.8	0.081
ImageDemoSobel	context	15.12	14.87	17.8	1.921
ImageDemoSobel	mispredict	15.82	15.01	17.2	2.011
Json	no-split	329.04	327.25	422.7	1
Json	default	289.8	288.43	385.9	0.881
Json	context	523.71	522.28	620.6	1.592
Json	mispredict	525.04	522.12	624.9	1.596
LiquidCartParse	no-split	77.61	77.09	78.7	1
LiquidCartParse	default	53.62	53.31	54.8	0.691

Table 22: Median execution time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
LiquidCartParse	context	89.63	88.83	90.8	1.155
LiquidCartParse	mispredict	71.99	71.75	73.9	0.928
LiquidCartRender	no-split	245.4	243.33	248.1	1
LiquidCartRender	default	75.54	74.61	76.9	0.308
LiquidCartRender	context	129.85	128.72	131.2	0.529
LiquidCartRender	mispredict	113.69	113.25	115.2	0.463
LiquidMiddleware	no-split	498.01	494.67	513.4	1
LiquidMiddleware	default	180.59	177.31	184.2	0.363
LiquidMiddleware	context	246.93	243.92	262.4	0.496
LiquidMiddleware	mispredict	214.56	211.5	228	0.431
LiquidParseAll	no-split	7294.49	7269.51	7304.8	1
LiquidParseAll	default	4931.25	4926.96	4979.1	0.676
LiquidParseAll	context	8063.02	8056.68	8071.4	1.105
LiquidParseAll	mispredict	6493.26	6486.99	6560.5	0.89
LiquidRenderBibs	no-split	19727.7	19687.93	19791.4	1
LiquidRenderBibs	default	4140.55	4131.13	4149.6	0.21
LiquidRenderBibs	context	9782.45	9768.47	9800.7	0.496
LiquidRenderBibs	mispredict	8040.46	8016.32	8076.1	0.408
List	no-split	75.78	75.27	160.4	1
List	default	75.86	75.31	160.6	1.001
List	context	237.34	236.66	324.6	3.132
List	mispredict	234.27	234.03	322	3.092
Mandelbrot	no-split	99.69	99.43	173.4	1
Mandelbrot	default	99.68	99.44	99.9	1
Mandelbrot	context	99.08	98.84	99.2	0.994
Mandelbrot	mispredict	99.09	98.84	176.6	0.994
MatrixMultiply	no-split	3059.85	3040.1	3114.6	1
MatrixMultiply	default	5204.75	5197.91	5208.7	1.701
MatrixMultiply	context	3986.06	3984.95	3987.7	1.303
MatrixMultiply	mispredict	3986.38	3985.52	3987.4	1.303

Table 22: Median execution time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
NBody	no-split	126.25	124.61	133.9	1
NBody	default	50.39	50.31	75.4	0.399
NBody	context	52.2	51.97	78.8	0.413
NBody	mispredict	52.13	51.97	77.2	0.413
OptCarrot	no-split	1426.59	1344.39	6127.9	1
OptCarrot	default	1355.47	1292.9	12914.8	0.95
OptCarrot	context	1512.71	1449.56	15361.5	1.06
OptCarrot	mispredict	1425.86	1357.37	13455.8	0.999
Permute	no-split	86.51	84.66	160.9	1
Permute	default	85.05	83.92	161.4	0.983
Permute	context	112.26	110.89	189.6	1.298
Permute	mispredict	112.01	111.21	189.5	1.295
PsychLoad	no-split	400.66	390.04	5448.1	1
PsychLoad	default	374.16	364.54	8102.8	0.934
PsychLoad	context	459.11	448.09	8697.1	1.146
PsychLoad	mispredict	443.79	434.75	8674.5	1.108
Queens	no-split	55.73	55.52	147	1
Queens	default	61.64	61.48	155.8	1.106
Queens	context	70.48	70.3	164.7	1.265
Queens	mispredict	70.5	70.22	167.1	1.265
RedBlack	no-split	859.04	849.23	1037.2	1
RedBlack	default	656.68	651.33	776.9	0.764
RedBlack	context	1030.5	1017.21	1196.5	1.2
RedBlack	mispredict	1027.73	1023.32	1197.4	1.196
Richards	no-split	78.28	78.19	275.1	1
Richards	default	70.05	69.99	271.2	0.895
Richards	context	306.41	305.29	630	3.914
Richards	mispredict	306.17	305.41	631.5	3.911
Sieve	no-split	406.41	405.2	491.1	1
Sieve	default	391.82	390.89	476.3	0.964

Table 22: Median execution time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
Sieve	context	406.69	406.05	495.5	1.001
Sieve	mispredict	406.86	406.02	494.5	1.001
Sinatra	no-split	142.69	141.67	227.5	1
Sinatra	default	72.91	72.07	156.9	0.511
Sinatra	context	104.07	103.27	191.3	0.729
Sinatra	mispredict	90.91	89.95	177.9	0.637
SpectralNorm	no-split	72.03	71.94	217.1	1
SpectralNorm	default	75.11	75.02	160.1	1.043
SpectralNorm	context	75.08	74.92	209.1	1.042
SpectralNorm	mispredict	75.29	75.2	162.2	1.045
Storage	no-split	81.63	81.12	165.8	1
Storage	default	81.96	81.22	167.4	1.004
Storage	context	121.68	120.68	199.8	1.491
Storage	mispredict	124.65	123.56	203.8	1.527
Towers	no-split	71.7	71.61	163.3	1
Towers	default	71.84	71.7	163.4	1.002
Towers	context	156.62	156.21	242.3	2.184
Towers	mispredict	155.43	155.22	240.1	2.168

B.4.2 Compilation time, startup excluded

Table 23: Median compilation time per iteration, startup excluded. All benchmarks, normalised to osplit.

Benchmark	VM	median	min	max	ratio
ADConvert	no-split	198	0	383	1
ADConvert	default	97	0	190	0.49
ADConvert	context	132	0	325	0.67
ADConvert	mispredict	134	0	266	0.68
ADLoadFile	no-split	107	0	303	1
ADLoadFile	default	99	0	250	0.93
ADLoadFile	context	99	0	300	0.93
ADLoadFile	mispredict	69	0	348	0.64
BinaryTrees	no-split	602	20	1083	1
BinaryTrees	default	637	19	1114	1.06
BinaryTrees	context	624	14	1101	1.04
BinaryTrees	mispredict	616	12	1088	1.02
BlogRails	no-split	48	1	86	1
BlogRails	default	12	0	142	0.25
BlogRails	context	52	0	53	1.09
BlogRails	mispredict	10	1	23	0.21
Bounce	no-split	1708	23	2011	1
Bounce	default	1262	54	1565	0.74
Bounce	context	1325	27	1636	0.78
Bounce	mispredict	1357	36	1650	0.79
CD	no-split	380	0	674	1
CD	default	391	0	684	1.03
CD	context	330	0	624	0.87
CD	mispredict	223	0	492	0.59
DeltaBlue	no-split	1122	4	1618	1
DeltaBlue	default	1308	6	1809	1.17
DeltaBlue	context	1096	29	1587	0.98
DeltaBlue	mispredict	1219	4	1681	1.09

Table 23: Median compilation time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	VM	median	min	max	ratio
FannkuchRedux	no-split	373	22	441	1
FannkuchRedux	default	522	14	599	1.4
FannkuchRedux	context	453	26	527	1.21
FannkuchRedux	mispredict	457	38	532	1.23
Havlak	no-split	391	0	598	1
Havlak	default	367	3	559	0.94
Havlak	context	395	0	586	1.01
Havlak	mispredict	338	0	567	0.86
HexaPdfSmall	no-split	893	1	1255	1
HexaPdfSmall	default	568	2	1226	0.64
HexaPdfSmall	context	1207	2	1643	1.35
HexaPdfSmall	mispredict	499	64	832	0.56
ImageDemoConv	no-split	491	30	865	1
ImageDemoConv	default	282	2	702	0.57
ImageDemoConv	context	429	2	819	0.87
ImageDemoConv	mispredict	448	2	829	0.91
ImageDemoSobel	no-split	351	12	669	1
ImageDemoSobel	default	65	4	76	0.19
ImageDemoSobel	context	537	2	955	1.53
ImageDemoSobel	mispredict	583	37	944	1.66
Json	no-split	572	4	941	1
Json	default	794	0	1126	1.39
Json	context	673	0	1055	1.18
Json	mispredict	730	0	1043	1.28
LiquidCartParse	no-split	402	4	586	1
LiquidCartParse	default	415	0	519	1.03
LiquidCartParse	context	446	4	582	1.11
LiquidCartParse	mispredict	358	0	461	0.89
LiquidCartRender	no-split	524	29	821	1
LiquidCartRender	default	285	1	549	0.54

Table 23: Median compilation time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	VM	median	min	max	ratio
LiquidCartRender	context	520	54	665	0.99
LiquidCartRender	mispredict	245	4	500	0.47
LiquidMiddleware	no-split	574	11	924	1
LiquidMiddleware	default	918	5	1134	1.6
LiquidMiddleware	context	683	11	921	1.19
LiquidMiddleware	mispredict	792	6	1014	1.38
LiquidParseAll	no-split	622	19	1051	1
LiquidParseAll	default	395	3	705	0.64
LiquidParseAll	context	527	2	804	0.85
LiquidParseAll	mispredict	430	3	732	0.69
LiquidRenderBibs	no-split	495	38	830	1
LiquidRenderBibs	default	244	0	563	0.49
LiquidRenderBibs	context	305	28	634	0.62
LiquidRenderBibs	mispredict	242	1	597	0.49
List	no-split	811	8	1120	1
List	default	817	10	1157	1.01
List	context	931	8	1265	1.15
List	mispredict	939	10	1326	1.16
Mandelbrot	no-split	1423	155	1826	1
Mandelbrot	default	208	0	310	0.15
Mandelbrot	context	216	0	318	0.15
Mandelbrot	mispredict	4144	113	4672	2.91
MatrixMultiply	no-split	785	24	1283	1
MatrixMultiply	default	506	0	831	0.64
MatrixMultiply	context	483	0	816	0.62
MatrixMultiply	mispredict	489	0	824	0.62
NBody	no-split	493	0	880	1
NBody	default	1003	14	1491	2.03
NBody	context	965	7	1348	1.96
NBody	mispredict	1010	22	1480	2.05

Table 23: Median compilation time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
OptCarrot	no-split	3133	33	4198	1
OptCarrot	default	3988	265	6186	1.27
OptCarrot	context	3743	8	4887	1.19
OptCarrot	mispredict	3392	21	4570	1.08
Permute	no-split	1271	9	1651	1
Permute	default	1127	16	1455	0.89
Permute	context	1103	8	1394	0.87
Permute	mispredict	1084	10	1407	0.85
PsychLoad	no-split	3676	56	4082	1
PsychLoad	default	12222	1840	12734	3.32
PsychLoad	context	2156	335	2701	0.59
PsychLoad	mispredict	9204	299	9633	2.5
Queens	no-split	996	0	1319	1
Queens	default	902	0	1178	0.91
Queens	context	1082	0	1361	1.09
Queens	mispredict	1058	7	1333	1.06
RedBlack	no-split	986	18	1273	1
RedBlack	default	1092	17	1358	1.11
RedBlack	context	1075	15	1336	1.09
RedBlack	mispredict	1097	16	1359	1.11
Richards	no-split	901	0	1342	1
Richards	default	978	0	1246	1.09
Richards	context	874	0	1152	0.97
Richards	mispredict	890	0	1179	0.99
Sieve	no-split	1250	6	2981	1
Sieve	default	1311	8	3009	1.05
Sieve	context	1560	8	3540	1.25
Sieve	mispredict	1346	11	3053	1.08
Sinatra	no-split	471	7	686	1
Sinatra	default	951	1	1157	2.02

Table 23: Median compilation time per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
Sinatra	context	984	6	1237	2.09
Sinatra	mispredict	1124	62	1332	2.39
SpectralNorm	no-split	1235	22	1805	1
SpectralNorm	default	856	0	1223	0.69
SpectralNorm	context	843	0	1330	0.68
SpectralNorm	mispredict	833	0	1195	0.67
Storage	no-split	963	1	1265	1
Storage	default	967	0	1275	1
Storage	context	960	8	1251	1
Storage	mispredict	964	5	1275	1
Towers	no-split	869	0	1280	1
Towers	default	894	0	1188	1.03
Towers	context	914	0	1211	1.05
Towers	mispredict	938	0	1231	1.08

B.4.3 Time spent in garbage collection, startup excluded

Table 24: Median time spent in garbage collection per iteration, startup excluded. All benchmarks, normalised to osplit.

Benchmark	VM	median	min	max	ratio
ADConvert	no-split	1098.5	11	2187	1
ADConvert	default	690	9	1371	0.628
ADConvert	context	764.5	8	1517	0.696
ADConvert	mispredict	584	8	1160	0.532
ADLoadFile	no-split	570	8	1122	1
ADLoadFile	default	260	3	523	0.456
ADLoadFile	context	440	16	852	0.772
ADLoadFile	mispredict	274.5	4	541	0.482
BinaryTrees	no-split	46	2	84	1
BinaryTrees	default	38	2	74	0.826
BinaryTrees	context	47	0	90	1.022
BinaryTrees	mispredict	55	0	97	1.196
BlogRails	no-split	85	9	160	1
BlogRails	default	88	8	163	1.035
BlogRails	context	86	6	164	1.012
BlogRails	mispredict	87	8	162	1.024
Bounce	no-split	309	8	594	1
Bounce	default	0	0	2	0
Bounce	context	0	0	0	0
Bounce	mispredict	0	0	0	0
CD	no-split	92	1	185	1
CD	default	79.5	1	157	0.864
CD	context	110	1	217	1.196
CD	mispredict	99.5	1	198	1.082
DeltaBlue	no-split	186.5	3	340	1
DeltaBlue	default	0	0	0	0
DeltaBlue	context	47.5	0	83	0.255
DeltaBlue	mispredict	15	0	28	0.08

Table 24: Median time spent in garbage collection per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
FannkuchRedux	no-split	0	0	0	0
FannkuchRedux	default	0	0	0	0
FannkuchRedux	context	0	0	0	0
FannkuchRedux	mispredict	0	0	0	0
Havlak	no-split	958	7	1921	1
Havlak	default	469.5	3	933	0.49
Havlak	context	655	4	1294	0.684
Havlak	mispredict	481.5	4	969	0.503
HexaPdfSmall	no-split	515	21	1036	1
HexaPdfSmall	default	399.5	18	823	0.776
HexaPdfSmall	context	508.5	17	1033	0.987
HexaPdfSmall	mispredict	512	16	1007	0.994
ImageDemoConv	no-split	22	1	43	1
ImageDemoConv	default	0	0	0	0
ImageDemoConv	context	183.5	3	356	8.341
ImageDemoConv	mispredict	173	2	343	7.864
ImageDemoSobel	no-split	0	0	2	0
ImageDemoSobel	default	0	0	0	0
ImageDemoSobel	context	0	0	0	0
ImageDemoSobel	mispredict	0	0	0	0
Json	no-split	384	4	814	1
Json	default	345	3	719	0.898
Json	context	455.5	5	962	1.186
Json	mispredict	459	4	969	1.195
LiquidCartParse	no-split	9	0	15	1
LiquidCartParse	default	3	0	5	0.333
LiquidCartParse	context	10	0	23	1.111
LiquidCartParse	mispredict	6	0	7	0.667
LiquidCartRender	no-split	30	0	49	1
LiquidCartRender	default	11	0	15	0.367

Table 24: Median time spent in garbage collection per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
LiquidCartRender	context	14	0	20	0.467
LiquidCartRender	mispredict	1	0	4	0.033
LiquidMiddleware	no-split	102	2	189	1
LiquidMiddleware	default	73	2	124	0.716
LiquidMiddleware	context	54	1	109	0.529
LiquidMiddleware	mispredict	25	0	40	0.245
LiquidParseAll	no-split	1748	25	3471	1
LiquidParseAll	default	1043	16	2060	0.597
LiquidParseAll	context	2178.5	30	4326	1.246
LiquidParseAll	mispredict	1051.5	16	2077	0.602
LiquidRenderBibs	no-split	2497	35	4969	1
LiquidRenderBibs	default	472.5	8	936	0.189
LiquidRenderBibs	context	1685	26	3339	0.675
LiquidRenderBibs	mispredict	665	11	1304	0.266
List	no-split	4	0	6	1
List	default	2	0	5	0.5
List	context	0	0	0	0
List	mispredict	6	0	10	1.5
Mandelbrot	no-split	0	0	0	0
Mandelbrot	default	0	0	0	0
Mandelbrot	context	0	0	0	0
Mandelbrot	mispredict	0	0	0	0
MatrixMultiply	no-split	28	1	52	1
MatrixMultiply	default	1477.5	29	2927	52.768
MatrixMultiply	context	25	0	53	0.893
MatrixMultiply	mispredict	25	1	52	0.893
NBody	no-split	37	0	70	1
NBody	default	0	0	0	0
NBody	context	0	0	0	0
NBody	mispredict	0	0	0	0

Table 24: Median time spent in garbage collection per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
OptCarrot	no-split	1133	20	2194	1
OptCarrot	default	1047.5	21	2035	0.925
OptCarrot	context	1181	21	2276	1.042
OptCarrot	mispredict	1029.5	16	1966	0.909
Permute	no-split	19	0	32	1
Permute	default	28	0	54	1.474
Permute	context	31	0	54	1.632
Permute	mispredict	34	2	57	1.789
PsychLoad	no-split	293	12	468	1
PsychLoad	default	384.5	29	547	1.312
PsychLoad	context	325.5	8	559	1.111
PsychLoad	mispredict	298.5	5	500	1.019
Queens	no-split	0	0	0	0
Queens	default	4	0	6	4000000
Queens	context	0	0	10	0
Queens	mispredict	3	0	5	3000000
RedBlack	no-split	67.5	2	124	1
RedBlack	default	15	0	16	0.222
RedBlack	context	2	0	3	0.03
RedBlack	mispredict	0	0	7	0
Richards	no-split	0	0	0	0
Richards	default	0	0	0	0
Richards	context	0	0	0	0
Richards	mispredict	0	0	0	0
Sieve	no-split	7.5	0	25	1
Sieve	default	18	5	33	2.4
Sieve	context	24	0	34	3.2
Sieve	mispredict	38	0	48	5.067
Sinatra	no-split	35.5	0	67	1
Sinatra	default	2	0	7	0.056

Table 24: Median time spent in garbage collection per iteration, startup excluded. All benchmarks, normalised *(continued)*

Benchmark	**VM**	**median**	**min**	**max**	**ratio**
Sinatra	context	19	0	25	0.535
Sinatra	mispredict	2	0	4	0.056
SpectralNorm	no-split	35	0	49	1
SpectralNorm	default	23	0	35	0.657
SpectralNorm	context	24	0	106	0.686
SpectralNorm	mispredict	22	0	39	0.629
Storage	no-split	14	0	28	1
Storage	default	20	0	35	1.429
Storage	context	22.5	0	42	1.607
Storage	mispredict	21	0	36	1.5
Towers	no-split	33	0	37	1
Towers	default	14	0	17	0.424
Towers	context	12.5	0	29	0.379
Towers	mispredict	13	0	26	0.394