SPECIALISING DYNAMIC TECHNIQUES FOR IMPLEMENTING THE RUBY PROGRAMMING LANGUAGE

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2015

By
Chris Seaton
School of Computer Science

Contents

Word Count: 53442

List of Listings

List of Tables

List of Figures

List of Abbreviations

ABI Application Binary Interface. 169

ACF Autocorrelation Function. 11, 88

ACID Atomic, Consistent, Isolated and Durable. 71

API Application Programming Interface. 8, 25, 26, 62, 64, 83, 101, 118, 132, 150, 151, 152, 154, 160, 161, 162, 169, 170, 171, 172, 174, 180, 182, 183, 186, 191

AST Abstract syntax tree. 11, 27, 51, 52, 53, 54, 55, 59, 60, 61, 62, 63, 64, 65, 66, 67, 71, 78, 106, 107, 110, 111, 116, 120, 121, 126, 128, 129, 131, 132, 133, 142, 154, 157, 167, 175, 185, 191, 195, 196

CLR Common Language Runtime. 64, 72

DSL Domain-Specific Language. 61

FFI Foreign Function Interface. 172

GC Garbage collector. 76, 146, 147, 149, 150, 158, 167, 171, 186

GDB The GNU Project Debugger. 117

IC Inline Cache. 54, 58, 60

IR Intermediate Representation. 11, 47, 56, 59, 61, 63, 65, 67, 68, 118, 167

IRB Interactive Ruby. 8, 119, 125, 191

JIT Just-in-time (compiler or compilation). 28, 64, 65, 67, 71, 109

JNI Java Native Interface. 172, 186

JSON JavaScript Object Notation. 25

JSR Java Specification Request. 158

JVM Java Virtual Machine. 27, 28, 29, 56, 61, 62, 64, 66, 67, 71, 72, 73, 78, 83, 101, 109, 116, 146, 147, 149, 150, 152, 155, 157, 158, 159, 160, 167, 168, 172, 173, 188

LLVM Low Level Virtual Machine. 28, 56, 65, 66, 67, 71, 72, 73, 103, 172, 173

MRI Matz's Ruby Interpreter. 11, 45, 47, 65, 66, 67, 68, 71, 81, 85, 93, 115, 116, 117, 118, 119, 135, 136, 139, 145, 146, 148, 169, 170, 172, 183, 184, 186, 188, 189, 191

MVC Model-View-Controller. 25, 36

ORM Object-Relational Mapping. 36

OSR On-Stack Replacement. 64

PIC Polymorphic Inline Cache. 11, 55, 57, 58, 60, 61, 103, 104, 110, 173, 175

PNG Portable Network Graphics. 37, 213

Rbx Rubinius. 68

REE Ruby Enterprise Edition. 71

REPL Read-eval-print loop. 119

REST Representational State Transfer. 25

SOM Simple Object Machine. 109

SQL System Query Language. 36

VM Virtual Machine. 28, 56, 64, 65, 71, 72, 99, 108, 120, 145, 147, 148, 149, 154, 161, 163, 164, 165, 167, 172, 188

YARV Yet Another Ruby Virtual Machine. 65, 68, 72, 170

Abstract

Specialising Dynamic Techniques for Implementing The Ruby Programming Language
Chris Seaton
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2015

The Ruby programming language is dynamically typed, uses dynamic and late bound dispatch for all operators, method calls and many control structures, and provides extensive metaprogramming and introspective tooling functionality. Unlike other languages where these features are available, in Ruby their use is not avoided and key parts of the Ruby ecosystem use them extensively, even for inner-loop operations. This makes a high-performance implementation of Ruby problematic. Existing implementations either do not attempt to dynamically optimise Ruby programs, or achieve relatively limited success in optimising Ruby programs containing these features.

One way that the community has worked around the limitations of existing Ruby implementations is to write extension modules in the C programming language. These are statically compiled and then dynamically linked into the Ruby implementation. Compared to equivalent Ruby, this C code is often more efficient for computationally intensive code. However the interface that these C extensions provides is defined by the non-optimising reference implementation of Ruby. Implementations which want to optimise by using different internal representations must do extensive copying to provide the same interface. This then limits the performance of the C extensions in those implementations.

This leaves Ruby in the difficult position where it is not only difficult to

implement the language efficiently, but the previous workaround for that problem, C extensions, also limits efforts to improve performance.

This thesis describes an implementation of the Ruby programming language which embraces the Ruby language and optimises specifically for Ruby as it is used in practice. It provides a high performance implementation of Ruby's dynamic features, at the same time as providing a high performance implementation of C extensions. The implementation provides a high level of compatibility with existing Ruby implementations and does not limit the available features in order to achieve high performance.

Common to all the techniques that are described in this thesis is the concept of specialisation. The conventional approach taken to optimise a dynamic language such as Ruby is to profile the program as it runs. Feedback from the profiling can then be used to specialise the program for the data and control flow it is actually experiencing. This thesis extends and advances that idea by specialising for conditions beyond normal data and control flow.

Programs that call a method, or lookup a variable or constant by dynamic name rather than literal syntax can be specialised for the dynamic name by generalising inline caches. Debugging and introspective tooling is implemented by specialising the code for debug conditions such as the presence of a breakpoint or an attached tracing tool. C extensions are interpreted and dynamically optimised rather than being statically compiled, and the interface which the C code is programmed against is provided as an abstraction over the underlying implementation which can then independently specialise.

The techniques developed in this thesis have a significant impact on performance of both synthetic benchmarks and kernels from real-world Ruby programs. The implementation of Ruby which has been developed achieves an order of magnitude or better increase in performance compared to the next-best implementation. In many cases the techniques are 'zero-overhead', in that the generated machine code is exactly the same for when the most dynamic features of Ruby are used, as when only static features are used.

Declaration

This thesis contains work that includes collaborations with other researchers and has already formed joint publications.

Chapter 5 contains work on metaprogramming for dynamic languages that led to similar results to the work of Stefan Marr at Inria. We combined our research for a single joint publication [73].

Chapter 6 contains work that initially supported a small set of debugging functionality for dynamic languages, and was generalised in work with Michael Van de Vanter and Michael Haupt at Oracle Labs [92].

Chapter 7 contains work on exposing safepoints to guest languages running on a virtual machine, with the implementation later improved in collaboration with Benoit Daloze at Johannes Kepler Universität, Linz. It was later applied to JavaScript by Daniele Bonetta at Oracle Labs [23].

Chapter 8 contains collaborative work with Matthias Grimmer at Johannes Kepler Universität, Linz that applied an existing implementation of the C programming language with the my implementation of Ruby and an interoperability technique developed by Matthias Grimmer to provide support for C extensions for the Ruby language [47, 46]. This collaborative work will also form part of the doctoral thesis of Matthias Grimmer.

Copyright

Acknowledgements

I thank my wife Maxine for her eternal love and support, and my daughter Rosalie for making life a joy.

I thank my supervisor Mikel Luján for introducing me to the academic world, his invaluable advice, and his amazing patience throughout this process.

I thank all the people I have authored papers with, including Daniele Bonetta, Benoit Daloze, Stéphane Ducasse, Matthias Grimmer, Michael Haupt, Christian Humer, Mikel Luján, Stefan Marr, Hanspeter Mössenböck, Michael Van de Vanter, Ian Watson, Christian Wirth, Andreas Wöss, and Thomas Würthinger.

I thank the other people at Oracle Labs I worked with in California and Linz including Mario Wolczko, Christian Wimmer and Peter Kessler, and the members of JRuby+Truffle team that we built as the project matured, including Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish and Lucas Allan. I'm proud to say that this list includes students, academics, industrial researchers and open-source collaborators.

This research was generously funded by a Doctoral Training Award from Her Majesty's Government, and an internship and later employment with Oracle Labs.

Chapter 1

Introduction

1.1 Dynamic Programming Languages

Informally, a *dynamic* programming language is one where more decisions must be taken as the program runs. The opposite of dynamic is *static*, and a static programming language is one where more decisions are taken as the program is compiled. This loose definition for dynamic can apply to various parts of the language in different ways. This is a spectrum and few languages are entirely static or entirely dynamic. Most languages have at least some features which can be described as dynamic.

A *dynamic type system* is one where type checking of a program is not performed until the code is executed. In dynamically typed languages, variable names in the source code are often not annotated with types and instead values carry information about their type, which is used at runtime to check typing requirements. More static languages may discard this type information at the compilation phase after statically checking typing requirements, or they may only retain limited information.

A *dynamically bound dispatch* system, also known as *late binding*, is one where for a given variable and a given method name, it is not possible to determine until runtime which method will be called. In many dynamic languages, the late binding is implemented as a lookup by name in the receiver object which is logically executed for every method call.

The related term *dynamic dispatch* refers to selecting at runtime which version of a method, having already determined which method to call. For example, resolving which implementation of the same method to call in a hierarchy created

by inheritance is an example of dynamic dispatch. Dynamic dispatch such as virtual calls in C++ are an example of a limited form of dynamic behaviour in a language generally considered to be static.

Metaprogramming can also be described as a dynamic language feature. Metaprogramming, and the related technique *reflection*, allows a program to examine and possibly modify the program, the data in it, and information about that data at runtime. A simple case is calling a method based on a name that is a value not present literally in the source code of the program. Some dynamic languages such as Ruby have powerful metaprogramming features that allow most operations to be executed as a meta-operation, supplying parameters that would normally be static, such as a method name, as runtime data.

Tooling and instrumentation is another language feature that can be seen as being on the dynamic spectrum. For example, whether or not a line of code is running with a line breakpoint attached can be treated as a dynamic property – not determinable at compile time and changing when the debugger is attached.

These terms used are, of course, not precise. As already stated, many languages have features that could be described as dynamic, as well as features that could be described as static. It is also the case that although in a static language all decisions may be made at compile time, but that does not mean that there were other decisions that could have been made, or could have been made better, at runtime. For example in the C programming language, which is on the very static end of the spectrum, a call through a function pointer is dynamic in the sense that the machine code location called is not in general determinable statically. However at runtime it may be possible to speculate on the function that the pointer references.

Another complication in terminology is that there is not necessarily a clear division between compile-time and runtime in many language implementations. There are pure interpreters, where there is no processing of code before it is executed, and pure compilers, where all code is generated ahead-of-time, but a great many language implementations do not fit into these categories. Even a very basic interpreter is likely to have a phase where source code is compiled to an internal bytecode, and even ahead-of-time compiled language may defer machine code generation until runtime.

In general, dynamic languages often offer less performance than static languages, because more decisions that must be taken to run the program means a

greater overhead. Due to dynamic type systems, values may have overhead in carrying about type information and types may have to be checked before operations can be performed. Due to dynamic binding, work may have to be done to decide which method to run for a given receiver and method name every time a call is made.

Even within the same language, more dynamic features often offer less performance than their static equivalents, for the same reason that extra work must be done. For example in Java a call using the `java.lang.reflection` API is around 6x slower than a method call written literally in the source code [73]. In many cases this means that these dynamic features are avoided in writing programs, or are used but then factored out when performance is poor.

These problems both encourage people to not use dynamic languages if they want performance, and to avoid using the more dynamic features of the language that they are using. As this thesis will describe, avoiding dynamic features can be in conflict with the preferred programming style of the language and the wider ecosystem, and means that programmers are not being well-served by the language implementers.

The characteristics of dynamic programming languages in the context of Ruby and the relevance of this to their optimisation is discussed in more depth in Chapter 2.

1.2 Idiomatic Ruby

Most of this thesis is concerned with the Ruby programming language. Ruby is a general purpose language, but it is probably most commonly used as a back-end web development language, meaning that is is used to serve web pages with content dynamically generated using a database, or to provide a web service API with technologies such as REST and JSON.

As a web development language, Ruby is often used with the Ruby on Rails framework [49]. Rails is an MVC web framework for creating web pages with dynamic content provided by a database backend or requests to web services. At various points in time, Rails has been a critical part of the infrastructure of billion dollar companies including Twitter, Shopify, GitHub and Hulu.

Rails is often called an 'opinionated' framework, in that in strongly encourages one way of writing web applications that they consider to be the best. Design decisions make that way as easy as possible, and other approaches may be much harder to implement, even intentionally so. As part of this philosophy, Rails favours convention-over-configuration and don't-repeat-yourself, meaning that the framework provides sensible defaults which you are normally expected to use without modification, and that repetitive boilerplate code is removed by not having to specify this configuration.

This approach to developing web applications obviously works very well for a great many people, but there is a trade-off. In order to provide a system that works without configuration and without boilerplate code, Rails makes very extensive use of Ruby's dynamic language features. Classes are constructed at runtime, objects are wrapped with proxies to intercept methods and modify behaviour, and methods are dynamically generated.

As Rails is so significant in the Ruby community, its approach to design has influenced the rest of the Ruby. Dynamic typing, dynamic binding and metaprogramming are frequently used even for basic operations that in most languages would be expressed statically. This programming style, preferring dynamic construction of systems, leads to a common style of *idiomatic* Ruby code that uses metaprogramming and other dynamic features extensively.

Another impact that this has had on the Ruby ecosystem is that high performance code has been rewritten in C in the form of dynamically loaded native extension modules, known as C extensions. Compared to the slowest implementation of Ruby, this does deliver a performance increase because it bypasses most of the dynamic features. However, long term it is a self-defeating strategy. C extensions are developed against a poorly defined API that is essentially the internals of the initial implementation of Ruby. Meeting this API at the same time as providing higher performance for Ruby code is highly problematic for newer implementations of Ruby. The problem of C extensions has grown as legacy code has accumulated. As in other communities such as Python, poor support for C extensions in alternative implementations of Ruby has limited their adoption.

The major implementations of Ruby are discussed in Chapter 3. The impact of Ruby culture and idiomatic Ruby code practices on optimisation is explained in Section 2.3.

1.3 Research Questions

The central research question that this thesis looks to answer is, *is it possible to optimise an implementation of the Ruby programming language for the language as it is idiomatically used, including its most dynamic features?*

- Can metaprogramming operations have the same performance as their static equivalents?
- Can dynamic languages have tooling such as debuggers that do not impact on the performance of the language when the tool is not being used, and have minimal impact until the tool is actually activated?
- Can powerful optimisations be applied in an implementation of Ruby without limiting the features that such an implementation supports?
- Can an alternative implementation of Ruby that works to achieve high performance also provide a high performance of C extensions?

1.4 Implementation Work

The majority of the experiments in this thesis were undertaken using a novel implementation of the Ruby programming language, JRuby+Truffle, implemented in the Java programming language using the Truffle framework for writing self-optimising AST interpreters, and the Graal dynamic compiler.

JRuby+Truffle began life as RubyTruffle, a standalone implementation of Ruby, using only the lexical analysis and parsing phases from the existing JRuby implementation [78]. JRuby is an implementation of Ruby also in Java, but using conventional JVM language implementation techniques such as bytecode generation and the `invokedynamic` instruction.

This was suitable for early experiments but in order to support more of the Ruby language and libraries, beyond the level of a simple demonstrator, it was necessary to re-use more code from JRuby. Therefore in early 2014 RubyTruffle was open-sourced and merged into the master branch of the JRuby code base as an optional backend and now known as JRuby+Truffle, where it is now part of production releases of JRuby.

JRuby+Truffle also reuses a significant volume of code from the Rubinius project [86]. Rubinius is an implementation of Ruby with a VM written in C++

and a dynamic compiler using the LLVM compiler framework. Some of the Ruby-specific features that are normally implemented in C or Java are implemented in Ruby in Rubinius, on top of an FFI and a system of primitive operations. By implementing the same FFI and primitives, JRuby+Truffle re-uses this Ruby code.

JRuby+Truffle is by far the fastest implementation of Ruby for a non-trivial set of benchmarks, often performing an order of magnitude faster than any other implementation on both synthetic benchmarks and kernels from real applications that people are using in production to make money today. JRuby+Truffle out-performs implementations using competing state-of-the art compilation techniques including the LLVM compiler, the JVM's `invokedynamic` instruction, and the RPython meta-tracing JIT compiler. This is while supporting compatibility with over 93% of the Ruby language and over 85% of the Ruby core library which is within around 10% of the level supported by other implementations (compatibility is discussed in Section 4.4). In some areas, such as support for the metaprogramming techniques discussed in Chapter 7, JRuby+Truffle's compatibility is better than the existing slower implementations.

JRuby+Truffle has now been used as a research foundation for other teams working on dynamic languages and VMs, such as in [76, 91, 105, 102].

The implementation of JRuby+Truffle is discussed in more depth in Section 3.3.5. Comparisons to JRuby and Rubinius are made in Subsection 3.3. Availability of source code of the implementation is detailed in Subsection 4.7.2.

1.5 Contributions

The main contribution of this thesis is an approach to language implementation that embraces and optimises the Ruby programming language as it is idiomatically used, tackling the most dynamic and most difficult to implement parts of the Ruby language rather than excluding them as has been the case in previous work.

- Chapter 5 contributes a technique for, and implementation of, inline caches generalised to optimise common metaprogramming operations. This reduces the cost of metaprogramming in most cases to be the same as using the equivalent static operations.

- Chapter 6 contributes a technique for, and implementation of, zero-overhead debugging. That is, a debugger which allows a line breakpoint to be attached with no significant impact on peak performance until the breakpoint is triggered. Conditional breakpoints have no significant impact on peak performance beyond that which writing the condition inline in the original source code would have.

- Chapter 7 contributes a technique for, and implementation of, safepoints for guest languages on the JVM that allows program execution to be paused to perform dynamic operations such as debugging, introspection and dynamic reconfiguration.

- Chapter 8 contributes a technique for, and implementation of, C extensions for Ruby that is compatible with high performance optimisations.

- Appendix A describes earlier work on a technique for parallelising applications where dependencies are dynamic and so, like in a dynamic language, cannot be statically analysed.

1.6 Publications

The work in this thesis is documented in the following peer-reviewed publications:

- S. Marr, **C. Seaton**, and S. Ducasse. Zero-Overhead Metaprogramming. In Proceedings of the 35th Conference on Programming Language Design and Implementation, 2015

- **C. Seaton**, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In Proceedings of the 8th Workshop on Dynamic Languages and Applications (DYLA), 2014

- B. Daloze, **C. Seaton**, D. Bonetta, and H. Mössenböck. Techniques and Applications for Guest-Language Safepoints. In Proceedings of the 10th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS), 2015

- M. Grimmer, **C. Seaton**, T. Würthinger, and H. Mössenböck. Dynamically composing languages in a modular way: supporting C extensions for

dynamic languages. In MODULARITY 2015: Proceedings of the 14th International Conference on Modularity, 2015

- M. Grimmer, R. Schatz, **C. Seaton**, T. Würthinger, and H. Mössenböck. Memory-safe Execution of C on a Java VM. In Workshop on Programming Languages and Analysis for Security, 2015

- M. Grimmer, **C. Seaton**, R. Schatz, T. Würthinger, and H. Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In Proceedings of the 11th Dynamic Languages Symposium, 2015

- **C. Seaton**, D. Goodman, M. Luján, and I. Watson. Applying Dataflow and Transactions to Lee Routing. In Proceedings of the 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), 2012 – **awarded best paper prize**

The author of this thesis also contributed to the following publications, which are not the primary subject of any chapters in this thesis.

- A. Wö, C. Wirth, D. Bonetta, **C. Seaton**, and C. Humer. An object storage model for the Truffle language implementation framework. In PPPJ '14: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools, 2014

- D. Goodman, S. Khan, **C. Seaton**, Y. Guskov, B. Khan, M. Luján, and I. Watson. DFScala: High Level Dataflow Support for Scala. In Data-Flow Execution Models for Extreme Scale Computing. IEEE, 2012

Blog posts on the author's website documented intermediate results:

- Deoptimising Ruby. What deoptimisation means for Ruby and how JRuby+Truffle implements and applies it. http://www.chrisseaton.com/rubytruffle/deoptimising/.

- Very High Performance C Extensions For JRuby+Truffle. How JRuby+Truffle supports C extensions. http://www.chrisseaton.com/rubytruffle/cext/

- Optimising Small Data Structures in JRuby+Truffle. Specialised optimisations for small arrays and hashes. http://www.chrisseaton.com/rubytruffle/small-data-structures/
- Pushing Pixels with JRuby+Truffle. Running real-world Ruby gems. http://www.chrisseaton.com/rubytruffle/pushing-pixels
- Tracing With Zero Overhead in JRuby+Truffle. How JRuby+Truffle implements set_trace_func with zero overhead, and how we use the same technique to implement debugging. http://www.chrisseaton.com/rubytruffle/set_trace_func/
- How Method Dispatch Works in JRuby/Truffle. How method calls work all the way from AST down to machine code. http://www.chrisseaton.com/rubytruffle/how-method-dispatch-works-in-jruby-truffle/
- A Truffle/Graal High Performance Backend for JRuby. Blog post announcing the open sourcing.
  http://www.chrisseaton.com/rubytruffle/announcement/

1.7 Thesis Structure

This thesis continues in Chapter 2 by describing the relevant characteristics of dynamic programming languages: what is meant by dynamism and why this impacts on their implementation. The language that is the subject of the research, Ruby, is introduced and is used to illustrate the relevant concepts.

Chapter 3 introduces the current techniques used to optimise dynamic programming languages, which leads into a description of the state-of-the-art techniques on which this thesis builds.

Chapter 4 explains the evaluation methodology that is used throughout the thesis, including the benchmarks and statistical methods applied. A scope is given within which results are evaluated and metrics for success are described and justified.

Chapter 5 begins coverage of the specific scientific contributions of this thesis, and explains how a dynamic language implementation can provide many important metaprogramming features with zero-overhead compared to equivalent, more static, features. Chapter 6 describes how debugging and introspection features

can be provided for a dynamic programming language, also with zero-overhead. Chapter 7 shows how an underlying virtual machine implementation technique, safepoints, can be re-used in the implementation of a dynamic language to provide advanced dynamic features in the Ruby language. Chapter 8 describes how dynamic languages are often extended using a static and natively compiled language and how the same dynamic techniques as used for the dynamic languages can be applied to also run the extensions. Chapter 9 concludes and describes potential future work.

Following the main body of the thesis, Appendix A describes additional work that describes a programming model for parallelising applications with dynamic parallelism that cannot be statically analysed. Appendix B gives some additional information on benchmarks introduced in Chapter 4.

A recommended reading path through the thesis depends on the interests of the reader. Chapter 2 give background information that explains the motivation for the work, but that a reader well-informed in the dynamic languages may already be aware of. Chapter 3 describes techniques which are applied throughout the thesis, and so is recommended prerequisite for all further chapters. Chapter 4 gives explanation of experimental methodology and so could be skipped unless the reader needs detailed understanding of the techniques applied in the evaluation sections of each contribution. Chapters 5 and 8, and Appendix A can be read independently, but it is recommended to read Chapter 6 before 7 as the latter builds on techniques from the former. Appendix A can be read separately to the other chapters. Appendix B provides extra information for readers who want more precise information on the benchmarks used.

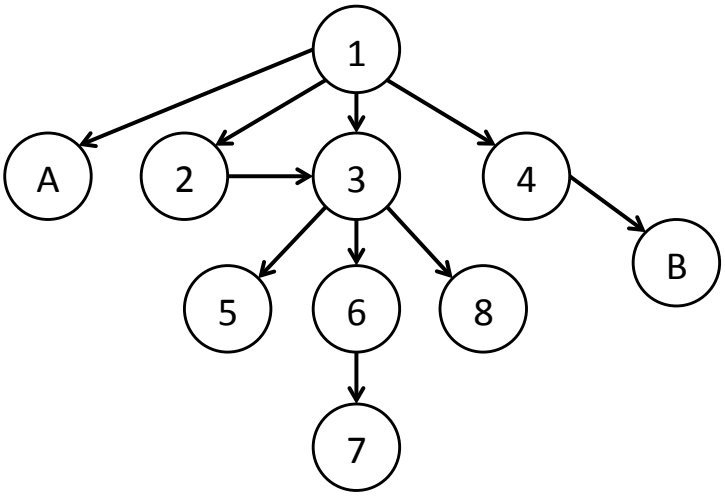


Figure 1.1: Recommended minimum chapter reading order

Chapter 2

Characteristics of Dynamic Languages

In this chapter we begin by introducing the programming language on which this thesis focuses, Ruby. We describe the key dynamic features of this language which later chapters will reference. We describe the framework which Ruby applications are probably most often written with, Rails. We then make a case study of idiomatic Ruby code and show how the dynamic features of Ruby are applied in key parts of the Ruby ecosystem and give examples of how their use can be found on performance critical paths.

2.1 Ruby

Ruby [116] is probably best described as an imperative, dynamically typed, dynamically and late bound programming language with extensive metaprogramming functionality [85]. Ruby has a heritage in Lisp and Smalltalk [74], and is broadly similar in look, feel and application to languages such as JavaScript and Python. A decade ago Ruby might have been described as a 'scripting language', but it is a general-purpose language in which many large enterprise applications are built which are critical to businesses' operations.

GitHub is a major example of a very large infrastructure built using Ruby [61]. Twitter was also built using Ruby until they decided to solve some scaling problems using Java and later Scala [35]. TravisCI, AirBnB, Hulu, Kickstarter are further examples of popular applications using Ruby.

Ruby is *dynamically typed* in that variables are not given explicit types by

the programmer, no types are statically inferred, and variables can hold objects of any type. Typing is not checked ahead of time or as the program is loaded. Type checks are made as the program runs and errors are reported in some cases or in others, values may be converted to meet typing requirements.

Ruby is *dynamically bound* or *late bound* in that method calls are logically resolved against the receiver object and based on the method name every time a method is called. For an arbitrary Ruby program it is not possible to determine what method will be called for a given call site in the program source code. The type receiver expression may not be known due to dynamic typing, and even if it were the set of methods in a class are mutable. For example, evaluating an argument in preparation for calling a method could redefine the method that is called, so the method must be logically looked up as the final action method before a call is made, and logically cannot be be moved in the program.

Ruby features extensive *metaprogramming* functionality such as the ability to call a method based on a name that is a dynamic runtime value, to evaluate code at runtime, to intercept method calls that would fail, to redefine methods, introspection of the environment and so on.

Ruby applications often make use of *C extensions* in order to work around the low performance of many Ruby implementations or to interoperate with libraries written in other language such as database drivers.

2.2 Ruby on Rails

Ruby on Rails, or just *Rails* [49], is the web-development framework that popularised Ruby in the western development community. It is an MVC framework and is famously *opinionated* in that it is designed to incorporate sensible defaults and does not go out of its way to encourage alternative configurations for most users.

Active Record is the ORM component of Rails, using metaprogramming to map database tables to classes, rows to objects and columns to methods. It allows a database to be accessed using objects instead of directly using SQL.

Active Support is a general utility library for Rails. It uses metaprogramming to add a large number of methods to the Ruby core library to make writing web applications more simple.

2.3 Case Study: Idiomatic Ruby

In this section we identify patterns that are common in Ruby that make extensive use of dynamic functionality. Many in the Ruby community accept these patterns as an acceptable way to write programs, valuing the expressiveness and flexibility they offer over more static functionality.

We use examples primarily from Rails and its component libraries Active Record and Active Support, as they are such key libraries for many Ruby applications. We also identify patterns in a pair of Ruby libraries, Chunky PNG [107] and PSD.rb [69]. These libraries are described in depth in Chapter 4, as kernels from them are key benchmarks for the work in this thesis. As a short introduction, Chunky PNG is a library for reading and writing PNG image format files, as well as editing them and managing data such as colour, and PSD.rb is a similar library for working with the Adobe Photoshop image format files.

Although these two libraries are not used anywhere near as commonly as Rails is, they provide clear additional examples of idiomatic Ruby patterns that we identify below. These libraries both have versions rewritten as C extensions, as operationally performance of the pure Ruby versions had been found to be too low. This gave us a clear need for a faster implementation of Ruby, and as only specific methods had been rewritten in C the areas which were too slow were also clear, and the kind of Ruby code they used could be studied.

2.3.1 Monkey Patching

Ruby modules and classes are always open for modification. There can be multiple definitions of a module or class, and if there are multiple definitions of a method then the last defined wins (in Ruby class and method definitions are executable statements so there will be an ordering). In the terminology of Liskov, Ruby modules and classes are both open for extension and open for modification. Adding and particularly overwriting existing methods is called *monkey patching* or sometimes *duck punching* (a reference to duck typing) in the Ruby community. The metaphor is one of mischief and indicates at the chaos this can potentially cause, but it is frequently found in important Ruby libraries.

An example of adding a method is in the Active Support component of Rails, shown in Listing 2.1. The `sum` method is added to the `Enumerable` mixin module, as Ruby does not provide one by default. The syntax to add additional methods

to an existing module is exactly the same as is used to create a new module, which indicates that adding additional methods after a first definition of the module is not considered a special case.

```
module Enumerable
  def sum(identity = 0, &block)
    if block_given?
      map(&block).sum(identity)
    else
      inject { |sum, element| sum + element } || identity
    end
  end
end
```

Listing 2.1: Active Support's adding the `sum` method to the existing `Enumerable` class

An example of overwriting a method can be found again in Active Support in Listing 2.2, which replaces the `DateTime.<=>` method for comparing two date values, with one that has special cases for the `Infinity` value. Again, the syntax here is exactly the same as for an original definition of a method. If a method with the same name already exists in the class, a subsequent definition will simply overwrite the old one. No special syntax is needed to do this and no warning is issued.

```
class DateTime
  # Layers additional behavior on DateTime#<=> so that Time and
  # ActiveSupport::TimeWithZone instances can be compared with a DateTime.
  def <=>(other)
    if other.kind_of?(Infinity)
      super
    elsif other.respond_to? :to_datetime
      super other.to_datetime rescue nil
    else
      nil
    end
  end
end
```

Listing 2.2: Active Support's overwriting the `<=>` method to the existing `DateTime` class

2.3.2 Dynamic Method Sends

The usual syntax to call a method in Ruby is the common `receiver.method(args)` notation, where the method name is a literal in the source code, not a dynamic value. Ruby also supports dynamic sends, similar to reflection in Java or metaobject protocols in other languages. In these, the name of the method is a dynamic value that can be produced from conditional statements, retrieved from a data structure, calculated from scratch or anything else you can do with a normal data value in Ruby.

Dynamic sends are often used where the method that needs to be called may depend on some logic, but the receiver and the arguments are always the same. Listing 2.3 is an example of this pattern. Line 4 uses a logical expression to select one of two symbols (immutable identifier objects, similar to interned strings), `:<` for less-than or `:<=` for less-than-or-equal-to. This is assigned to a local variable. Line 5 then uses a dynamic send, with the name of the method to call being the symbol that was chosen on the previous line. The argument is the same (the `last` attribute) for a call to either operator, so the programmer here has factored out the only part of the logic which varies, the name of the method. If conventional wisdom that metaprogramming is slow is put aside, this decision is in accordance with good software engineering practices, with the common parts of the code factored out and the part that varies isolated.

```
1  module IncludeWithRange
2    def include?(value)
3      if value.is_a?(::Range)
4        operator = exclude_end? && !value.exclude_end? ? :< : :<=
5        super(value.first) && value.last.send(operator, last)
6      else
7        super
8      end
9    end
10 end
```

Listing 2.3: Active Support's `IncludeWithRange` using a metaprogramming send

This particular example really shows how much Ruby developers value the elegance afforded by metaprogramming, as this method can be rewritten with just a little extra complexity to use conventional calls instead of dynamic sends,

as shown in Listing 2.4. This code is not accidentally written with metaprogramming by a developer who does not understand the cost in most implementations. The Rails code base is actively maintained with careful documentation and neat formatting, and subject to code review.

```
if exclude_end? && !value.exclude_end?
  super(value.first) && value.last < last
else
  super(value.first) && value <= last
end
```

Listing 2.4: `include?` from Listing 2.3 rewritten to use conventional calls

Listing 2.5 is an example of a related metaprogramming operation, `respond_to?`. This returns a value indicating whether or not an object responds to a message (has or inherits a method with that name which we can call). Here it is used to check that a call will succeed before the call is made. The actual call is made using a conventional send.

```
class Object
  # An object is blank if it's false, empty, or a whitespace string.
  # For example, '', '   ', +nil+, [], and {} are all blank.
  def blank?
    respond_to?(:empty?) ? !!empty? : !self
  end
end
```

Listing 2.5: Active Support using introspection on an object before attempting a call which may fail

We can see an even more extreme example of a dynamic send in the Chunky PNG routine for determining the colour that is used to represent transparent from a greyscale palette in an indexed image. The value needs to be interpreted in different encodings depending on the bit depth of the image. There are different routines such as `decode_png_resample_8bit_value` and `decode_png_resample_16bit_value`. Listing 2.6 shows how the name of the correct method is created through string interpolation with the value of current bit depth.

This is another example of where code duplication has been reduced by using a dynamic send. It would have been possible to lookup the correct method based

```
1 def grayscale_entry(bit_depth)
2   value = ChunkyPNG::Canvas.send(
3     :"decode_png_resample_#{bit_depth}bit_value",
4     content.unpack('n')[0])
5   ChunkyPNG::Color.grayscale(value)
6 end
```

Listing 2.6: Chunky PNG routine using send to choose a method based on input

on a large conditional expression on the bit depth variable, and it would have also been possible to factor that out into a helper method if desired, but the programmers here have valued the simplicity of expressing the method to call as a dynamically calculated name.

2.3.3 Dynamic Proxies and Forwarding

Another application of dynamic sends is to create a dynamic proxy of an object. A proxy is an object that wraps another object and when it receives a method call it sends it on to the wrapped object. The proxy may filter or modify these calls in some way to change the behaviour from the wrapped object. A reduced case of the proxy pattern is simple method forwarding from one object to another, without an explicit wrapped object.

Listing 2.7 shows an interesting use case where a method is forwarded from PSD.rb. The `hard_mix` method is an implementation of one technique for composing two image layers, that is found in the Photoshop file format. It takes foreground and background colours from the two layers and returns the composed colour that results from their composition. The method calls `r`, `g`, `b` methods on lines 10 to 15 to extract the channel components of the colours, but these methods are not defined in this module. Instead, there is a `method_missing` handler on line 19 which is run when a method is called but no such method is found in the object. It checks if the `ChunkyPNG::Color` module responds to that method instead on line 21. It it does respond to it, the method is called using a send on line 20 (conditions can be written in Ruby with the condition after the body, as in Perl), implementing a kind of dynamic namespace importing of these methods. The most interesting thing about this method is that it is called once per pixel when composing layers with this mode, so for large images it is extremely important for performance, and it is an example of metaprogramming

in the innermost loop of compute-bound production code.

```
def hard_mix(fg, bg, opts={})
  return apply_opacity(fg, opts)
    if fully_transparent?(bg)

  return bg if fully_transparent?(fg)

  mix_alpha, dst_alpha = calculate_alphas(
    fg, bg, DEFAULT_OPTS.merge(opts))

  new_r = blend_channel(r(bg), (r(bg)
    + r(fg) <= 255) ? 0 : 255, mix_alpha)
  new_g = blend_channel(g(bg), (g(bg)
    + g(fg) <= 255) ? 0 : 255, mix_alpha)
  new_b = blend_channel(b(bg), (b(bg)
    + b(fg) <= 255) ? 0 : 255, mix_alpha)

  rgba(new_r, new_g, new_b, dst_alpha)
end

def method_missing(method, *args, &block)
  return ChunkyPNG::Color.send(method, *args)
    if ChunkyPNG::Color.respond_to?(method)
  normal(*args)
end
```

Listing 2.7: PSD.rb forwarding methods using metaprogramming

An example of a more explicit proxy with a wrapped object is the `Duration` class in Active Support, shown in Listing 2.8. This class wraps a `Time` value, which is accessed by the `value` attribute defined on line 2, and adds methods such as `as_json` (not shown) and modifying other methods such as `inspect` (also not shown) to show the time as time period rather than an absolute point in time. All other methods are just forwarded to the wrapped `Time` value, via the dynamic send on line 17.

2.3.4 Dynamic Code Generation

Ruby programs often use dynamic code generation to extend the program at runtime. In some cases this is done to avoid the overhead of metaprogramming

```
1  class Duration
2    attr_accessor :value
3
4    def initialize(value)
5      @value = value
6    end
7
8    def as_json
9      ...
10   end
11
12   def inspect
13     ...
14   end
15
16   def method_missing(method, *args, &block)
17     value.send(method, *args, &block)
18   end
19 end
```

Listing 2.8: Active Support's `Duration` forwarding method calls to `Time`

in current implementations. Instead of using a dynamic send, a new conventional send is created at runtime by creating a string of Ruby code and evaluating it. As method definitions in Ruby are normal, imperative, executable statements, this can be done at any point in the program. It is also an application of monkey patching.

Active Support includes a class for writing delegates, and can generate a forwarding method as shown in Listing 2.9. `method_def` creates a string from several lines of Ruby code. The name of the method is inserted via string interpolation (the `"##{foo}"` syntax in Ruby is a string literal with the expression in brackets evaluated, converted to a string and inserted). Line 7 then evaluates that string as Ruby code, in the context of the current module. The method definition runs and adds the method to the module. The resulting method has a conventional call site with the name of the method inserted by interpolation when the code is generated, not when it is run. The advantage of this method is that it compensates for the poor implementation of `send` in other implementations of Ruby, but the downsides are that this complicates debugging (debuggers may find it hard to visualise stepping into code created from a dynamic string). It could also be

described as less readable, as code is now in a string literal, defeating tools such as syntax highlighters or the limited static analysis that is sometimes attempted on Ruby programs.

```
def delegate(method)
  method_def = (
    "def #{method}(*args, &block)\n" +
    "  delegated.#{method}(*args, &block)\n" +
    "end"
  )
  module_eval(method_def, file, line)
end
```

Listing 2.9: Active Support's `delegate` method dynamically creating a method to avoid metaprogramming (simplified)

Listing 2.10 shows another instance of the same pattern, this time from Active Record where methods are created to read and write fields that are dynamically configured from a database schema. In this case an effort has been made with the references to `__FILE__` and `__LINE__` to give the generated code proper source location information for error messages and debuggers, which indicates that this problem is known about. An alternate string literal syntax (the `<<-STR` and `STR` delimiters are effectively alternate double quotes here).

```
def define_method_attribute(name)
  generated_attribute_methods.module_eval <<-STR, __FILE__, __LINE__ + 1
    def #{temp_method}
      name = ::ActiveRecord::AttributeMethods::AttrNames::ATTR_#{name}
      _read_attribute(name) { |n| missing_attribute(n, caller) }
    end
  STR
end
```

Listing 2.10: Active Record's `define_method_attribute` method dynamically creating a getter for a database column (simplified)

For these cases we would like the programmer to be able to use dynamic sends rather than using code generation, if the only reason the programmer is avoiding metaprogramming is because of the cost in current implementations.

2.3.5 C Extensions

Both Chunky PNG and PSD.rb have optional C extension libraries that replace key methods in the libraries with native code. These are called Oily PNG and PSD Native. These extensions monkey-patch methods (which is Ruby's terminology for redefining a method while a program runs) in the original, pure Ruby, versions of the libraries.

For example, the Ruby version of the PSD.rb utility method `clamp`, which clamps a value between two bounds, uses high level constructs – an array, a sort operation and indexing:

```
1 def clamp(num, min, max)
2   [min, num, max].sort[1]
3 end
```

Listing 2.11: PSD.rb's `clamp` in pure Ruby

The equivalent C code does the same task but unlike the Ruby code it does not create an array or call Ruby's `sort` method. Instead it uses the Ruby C API to convert the objects to C primitive types and uses primitive C operators on them to clamp the value. This bypasses the dynamic nature of Ruby, and will normally achieve much higher performance.

```
1 VALUE psd_native_util_clamp(VALUE self,
2     VALUE r_num, VALUE r_min, VALUE r_max) {
3   int num = FIX2INT(r_num);
4   int min = FIX2INT(r_min);
5   int max = FIX2INT(r_max);
6   return num > max ?
7       r_max
8     : (num < min ? r_min : r_num);
9 }
```

Listing 2.12: PSD Native's `clamp` in C using the Ruby extension API

In the case of Chunky PNG and PSD.rb the authors have written a pure Ruby version of everything that they have written a C extension version for. This is ideal, because it means that slower implementations of Ruby such as MRI and Rubinius can run the C extension to improve performance, but an implementation with better performance may be able to run the Ruby code and

get high performance anyway. Not all C extensions are written in this way, and in some cases functionality is only implemented in C code. In this case, the high performance implementation of Ruby will need to be able to run the C code, as there is no Ruby code to run.

2.3.6 An Acid Test

The benchmarks and evaluation methodology used in this thesis are described in depth in Chapter 4, but we will pause here to give a concrete demonstration of the effect on performance of some of the patterns identified above.

We isolated some of the key problematic parts of the PSD.rb kernels in a synthetic benchmark, shown in Listing 2.13.

The benchmark defines a module, `Foo` with a method `foo` on line 2 which accepts three arguments. The method creates a `Hash` on line 3 with the three arguments, maps the hash's key value pairs to just the values to produce an `Array` on line 4. It takes the middle value of this array and assigns it to the temporary value `x` on line 5. The method then creates an array of the three arguments, sorts them and takes the middle value, assigning it to `y` on line 6. The two temporary values are added together and returned on line 7.

A class, `Bar`, has no methods but does define `Bar.method_missing` on line 12 which is called if a method is called on the class but not found. It checks if the `Foo` module responds to the method on line 13, and if so uses `Foo.send` to call the method on line 14.

The benchmark creates an instance of `Bar` on line 21, and calls `bar.foo` on it on line 27 with three constant arguments, in a loop, measuring the time for a million iterations.

In a sufficiently advanced implementation of Ruby, we would expect that the inner loop of this benchmark would be compiled to a constant value, as given this as the whole program there is no way for the result to be anything else. Dynamic functionality of Ruby such as monkey patching means that it is possible that methods or operators used in the benchmark may be redefined, but in this case they are not and the program will be stable. Achieving this requires deep inlining, including through metaprogramming sends, escape analysis of non-trivial data structures including hashes, compile time evaluation of library methods, including higher order methods such as `map`, optimistic assumption that methods are not redefined, and constant folding.

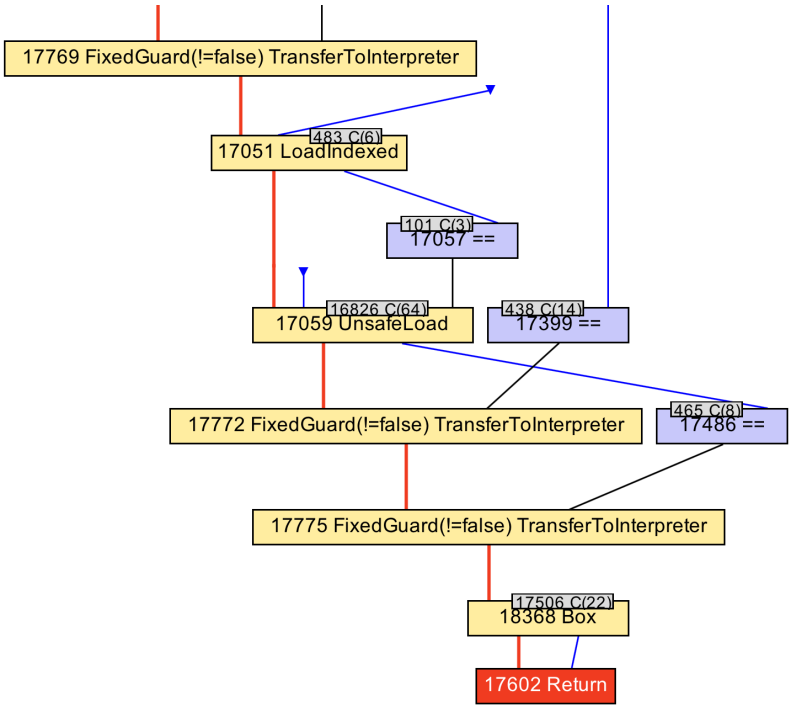


Figure 2.1: A sub-view of IR resulting from Listing 2.13

Currently, JRuby+Truffle is the only implementation that is able to reduce this benchmark to a constant value. We can verify that JRuby+Truffle can do this in two ways.

We can examine the IR using the Ideal Graph Visualiser tool [113]. Figure 2.1 shows a sub-view of the IR that results from running the Acid Test benchmark. The *return* node has a single data input (the blue edge), which is the boxed constant value 22 - the correct folded result. There is also a control dependency from the return node (the red edge), which leads to guards which check that assumptions taken to produce that constant value still hold.

We can also examine the produced machine code. Listing 2.14 shows a sub-view of a disassembly of the AMD64 machine code that results from running the Acid Test benchmark. As with the IR, there is code resulting from guards (631 bytes), but the method ends with loading the cached boxed representation of 22, and then returns.

We can also measure the performance of JRuby+Truffle for this benchmark compared to other implementations. Figure 2.2 shows speedup for this benchmark for other implementations of Ruby. JRuby and Topaz achieve up to around 3× the performance of MRI, but Rubinius is slower than MRI. Figure 2.3 shows

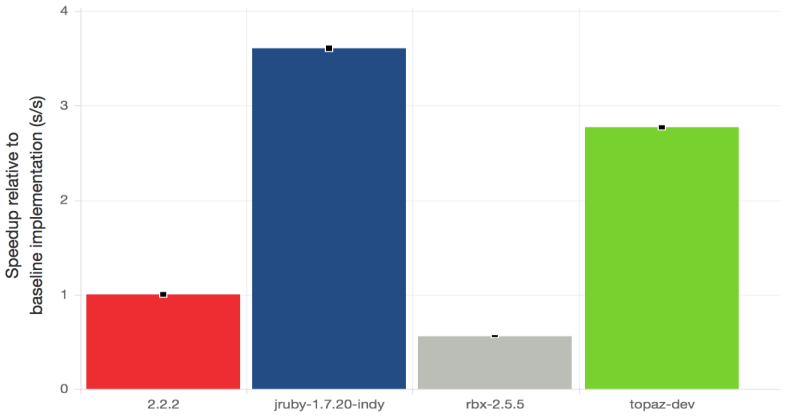


Figure 2.2: Performance of the Acid Test benchmark in existing implementations of Ruby

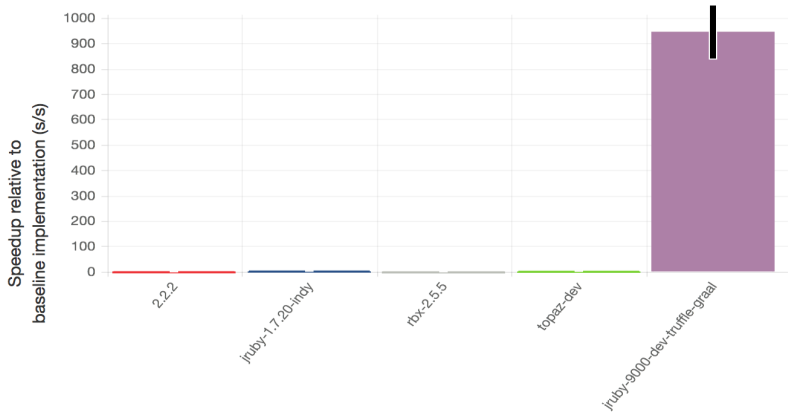


Figure 2.3: Performance of the Acid Test benchmark in JRuby+Truffle

the speedup of JRuby+Truffle in comparison to other implementations. As JRuby+Truffle is the only implementation of Ruby able to remove the indirection of all the metaprogramming and reduce the benchmark to a constant value, it is able to achieve performance around three orders of magnitude higher than MRI or any other implementation of Ruby. This benchmark give an early indication that JRuby+Truffle can optimise the patterns that we find in idiomatic Ruby, and what the impact of those optimisations is on performance.

Some of the techniques used to achieve this result, and a discussion of why other implementations are not as capable, can be found in Chapter 4. Chapter 5 covers the novel techniques which are most important for optimisation of this particular benchmark and those from which it was derived.

2.4 Summary

A great deal of the existing literature on Ruby assumes that features such as metaprogramming and C extensions are unimportant niche features which can be safely disregarded in analysis and implementation of Ruby programs [9, 10, 24, 29, 33, 32, 71]. The common idiomatic patterns which we have identified in core parts of the Ruby ecosystem show that this is not the case.

Existing workarounds for low performance of metaprogramming and other dynamic features of the Ruby language such as C extensions have created a situation where there is now legacy code written in C instead of Ruby. An implementation of Ruby must be able to run this C code, even if the problems which prompted them to be written have now been solved.

In this thesis we will describe our optimised implementation of Ruby where these features are optimised in the same way as conventional programming features. In performing this optimisation we do not sacrifice other aspects of Ruby such as debugging and tooling, and support for C extensions.

```ruby
module Foo
  def self.foo(a, b, c)
    hash = {a: a, b: b, c: c}
    array = hash.map { |k, v| v }
    x = array[0]
    y = [a, b, c].sort[1]
    x + y
  end
end

class Bar
  def method_missing(method, *args)
    if Foo.respond_to?(method)
      Foo.send(method, *args)
    else
      0
    end
  end
end

bar = Bar.new

loop do
  start = Time.now
  1_000_000.times do
    # This block should be compiled to the constant Fixnum value 22
    bar.foo(14, 8, 6)
  end
  puts Time.now - start
end
```

Listing 2.13: A benchmark indicative of patterns found in PSD.rb

```
...
movabs 0x11e2037a8, %rax ; {oop(a 'java/lang/Integer' = 22)}
...
retq
```

Listing 2.14: A sub-view of machine code resulting from Listing 2.13

Chapter 3

Implementation of Dynamic Languages

In this chapter we discuss some of the techniques which are used to create high performance implementations of dynamic programming languages. We begin with basic techniques which are widely applied, using Ruby as an example language to illustrate them, and then describe the specific techniques which are used in the implementation of the contributions of this thesis. We introduce the novel implementation of Ruby that is the subject of this thesis, JRuby+Truffle, and then describe the other existing implementations of Ruby which we will be comparing against. We then provide a broad overview comparison of the performance of these implementations of Ruby to set the context for the evaluation in the contribution chapters.

3.1 Foundational Techniques

3.1.1 Lexical and Syntactical Analysis

Lexical and syntactical analysis for dynamic languages are usually conventional, using a lexer and parser generator such as Flex and Bison. The output of syntactical analysis is usually an AST, which is the point at which the work in this thesis becomes relevant.

3.1.2 From ASTs to bytecode

An AST represents the source code of methods, functions or other such units of a program as a tree of operations. Each node in the AST is some named compute operation, with child nodes being either a control or data dependency for that operation. Additional data dependencies may exist in for example access to local variables which are not represented by edges in an AST, and by the sequential ordering of child nodes. Edges can be conditional based on the logic in the nodes to represent logic such as conditional statements, and child nodes can be executed repeatedly to represent logic such as loops.

To implement an AST in an object oriented language such as Ruby, Java or C++, there is usually a class for each type of node, and child nodes are connected via a pointer or reference. The compute operation can be implemented as an `execute` method in the class. An abstract base class or interface with a generic `execute` method can abstract across all node classes.

ASTs are *abstract* in the sense that some of the concrete syntactical elements found in the source code such as parentheses are not present in the AST. An AST may also use a range of levels of abstraction. For example, in Ruby most operators are actually a form of method call. A Ruby AST may represent operators explicitly, with a special class of node for operators, or it may represent operators using the same nodes as are used for method calls.

An AST can be directly executed by walking the tree from the root node, executing the compute action of each node in turn. The host language's stack is used to implicitly keep track of which node is currently executing and which node to return to when it is finished. Whether or not a child node is executed or executed repeatedly may depend on code in the compute action of the parent node. Some languages have features which may complicate this simple model, such as `goto` in C which may require a jump from one node to another.

Executing an AST by walking it is a conceptually simple way to run a program, but it is not often an efficient technique. If the AST is represented simply as a tree of objects then navigating the tree requires a lot of pointer or reference indirections which may not make optimal use of memory cache. Another overhead is the cost of making a method call to perform each compute action, and costs such as boxing needed to provide an abstract interface.

Many implementations of programming languages begin by using this kind of AST interpreter technique, due to the simplicity of implementation. When the

overheads become a problem, an option is to convert the AST that the parser gives you into a bytecode format.

Bytecode is an instruction set for a virtual machine. The program is represented as a linear set of instructions, similar to machine code but often at a higher level of abstraction. While a machine code instruction may perform some low-level action such as reading a memory location or native word arithmetic, a bytecode instruction may perform a much higher level action such as indexing a dynamically sized array with bounds checking. Nodes from the AST become instructions, with child nodes placed in the stream before parent nodes. The sequence of the instructions represents both the control and dataflow flow of the program. Jumps from one point in the stream to another allow for conditional execution of code or repeated execution.

Bytecode is usually executed by maintaining an instruction pointer that references the current instruction, and running a loop which takes the next instruction and performs the compute action. The body of the loop will often be a large switch statement on some operation code. Instead of data being passed along edges as in an AST, a bytecode format may use a stack where dependencies are popped off and results pushed on, or may use named or numbered registers that are referenced in the instruction stream [94].

More advanced techniques for executing bytecode include using a *threaded interpreter* where instructions are encoded as runtime procedure calls. In a *direct* threaded interpreter the instructions are literally the addresses of runtime routines, and the interpreter loop just runs each call in turn. In an *indirect* threaded interpreter the instructions are indices into a table of runtime routines which the interpreter loop looks up before making each call. In threaded interpreters data is usually passed on a stack, as there is no convenient place to store the registers to be used for each instruction. It would also be possible to have instructions that are objects with an `execute` method, as with an AST interpreter, but as the goal of bytecode is usually to have a more efficient representation of the program this is an unlikely technique.

3.1.3 Method Caching

As was shown in Chapter 2, dynamic and late method binding and metaprogramming sends are an essential feature of Ruby that is used on the hot path of key parts of the ecosystem. As all Ruby values are objects and most Ruby

operators are implemented as method calls on those objects, a Ruby program usually does very little but manipulate local variables and make method calls. The performance of method calls is therefore essential for good overall performance.

Chapter 2 previously showed that in Ruby the type of the receiver of a call cannot in general be statically determined, and that the set of methods defined on a class can change at any point, but we can make the reasonable assumption that they will most likely be stable. We can use this assumption to look up the method once, which is the expensive operation, and then *cache* a mapping from the tuple of the type and the method name to the correct method, in some shared global data structure like a hash map. The next time we execute a method call, if the type of the receiver and the method name is the same as before then we can lookup in the cache and use the cached method instead of looking up the correct method from scratch. If the set of methods in a class changes, then we need to clear the cache. The cache can be cleared globally, or it may be possible to selectively clear it based on a particular name or class.

This *global method cache* is the simplest form of method cache, but has several limitations. Using the cache still involves looking up in a map, so although it is useful in languages like Ruby which have very complicated method resolution rules (with inheritance, mixins etc), if the resolution algorithm was simple anyway then switching to looking up in a cache instead might not result in higher performance. The global method cache is also limited in size, and the size will probably not be related to the number of call sites in the program. As programs grow in complexity it is possible that the cache may be too small for the number of method calls in the hot path of the application, and the cache entries will be continually cleared and added.

A more advanced technique is to store a single-entry method cache with the instruction, as an *inline cache* (IC) [25]. In the case of an AST interpreter the cache can be stored in the node objects. In a bytecode interpreter, the instructions can be extended with a mutable field to store the cache. This is known as *bytecode quickening* [19]. The advantage of an inline cache is that you automatically scale the number of caches with the number of call sites. Disadvantages include that method resolution needs to be performed to fill each cache, rather than once for each tuple, which can be mitigated by pairing with a global cache that is used when filling the inline caches. Another disadvantage is that to clear the caches you now have to find many inline caches. This can be solved using a cache version

number, which can be incremented to say that all caches with a previous version are invalid. The caches can then be cleared lazily the next time they are accessed and the version number is found to be out of date with the latest version.

Another disadvantage of inline caches is that they store only one value. A single call site in most dynamic languages can see multiple receiver types, so a cache that stores a single value could be continually missed and refilled. A solution to this is a *polymorphic inline cache* (PIC), which stores multiple entries [54]. Unlike a global cache, a PIC is normally searched linearly rather than searched by hashing. The test that is applied to check that an entry in the cache is suitable is often called the *guard*.

With PICs there needs to be some mechanism to limit the size of the cache so it does not grow beyond the point where the linear search is effective and memory consumption is reasonable. A maximum size can be enforced, and beyond that size the cache is described as *megamorphic*. The cache could be removed and full method resolution used instead, or full resolution could be added to the end of the PIC, keeping the previous entries.

3.1.4 Dynamic Optimisation

As already described, AST interpreters have considerable overhead due to method calls and a non-linear data structure. Bytecode interpreters also have an overhead in the instruction loop, the switch statement for simple interpreters or the calls for threaded interpreters. Bytecode instructions also run independently of each other, with little possibility to optimise across them. An option to further increase performance is to *dynamically optimise*, or *just-in-time compile* the program to machine code [25].

A simple case form of dynamic optimisation is a template compiler, where each instruction is translated to a template of machine code. Like in a bytecode interpreter, data flow between these blocks of instructions may use a stack or some other standard interface. The performance of template compilers may be limited because optimisations between instructions are not considered, and each block of machine instructions are emitted independently of the next.

More advanced dynamic optimisation may use an advanced *intermediate representation* and multiple optimisation phases, sophisticated register allocation and instruction selection and scheduling, just like a powerful static compiler does, although the techniques may be tuned to reduce the time they consume, given

that they may be competing with the application for processor time. This allows the resulting machine code to be optimised across whole methods, just as a static compiler would.

Another option is to use a compiler framework such as LLVM, or re-use an existing VM that already uses dynamic optimisation such as the JVM. Here instead of emitting native machine instructions, the IR or bytecode of the framework or VM is used instead. This means that the advanced parts of a compiler are provided for you, but the disadvantage is that they are not specialised for your language. The best performing implementations of highly dynamic languages such as Google's V8 implementation of JavaScript often use finely specialised compiler phases, and V8 has hand-written assembly templates for some key operations. This isn't easy to add when you have added a level of abstraction between your implementation and the code generation.

Part of dynamic optimisation may be to gather profiling information while the interpreter runs, and use this information to specialise the code produced by the dynamic compiler for the program as it is actually running. Examples of profiling information that could be gathered include counts of which side of a branch is taken, what types are seen in variables, whether or not arithmetic overflows, and so on. Gathering this data will likely slow down the interpreter, so it trades-off cold-performance and time to warm-up with peak performance. This is a trade-off that is discussed further in Chapter 4 and is considered fair for our purposes.

3.1.5 Dynamic Deoptimisation

Dynamic optimisation can be made more effective with the ability to *dynamically deoptimise*. That is, to go from the optimised machine code back to the interpreter and resume running from the same point in the program but now running in the interpreter [55]. This is more complex than simply jumping from machine code into the interpreter, as the native code will likely use data structures such as stack activation frames in a different format than the interpreter, and if so these will need to be converted.

When machine code is deoptimised it may also be discarded so that it is not used again if it has become invalid. Dynamic deoptimisation without discarding the generated machine code can be called a *transfer to interpreter* and is useful where the machine code does not handle some behaviour but there is no

invalidation required and the same machine can be run next time.

If a language implementation is able to repeatedly dynamically optimise and deoptimise then the compiler can make aggressive speculative optimisations, always knowing that it can back-out to the interpreter and start again if the speculative assumptions do not hold.

One example of this is that a PIC can be compiled into the machine code so that it becomes a series of linear guards in the machine code. This means that there is no separate data structure to represent the cache that has to be fetched separately, and should improve cache efficiency. However the PIC is mutable and may need to change if the cache misses and a new entry is added, or if the cache transitions to a megamorphic configuration. One way to handle this is to always compile a fallback case at the end of the cache, but if the cache becomes invalid and the machine code cannot be discarded the cache will continue to miss and the fallback case will be used every time. With dynamic deoptimisation the program transfers to the interpreter where the cache is modified. New machine code with the modified cache can be produced again in the future.

Another example of the use of dynamic deoptimisation is removing branches that have not been executed in the interpreter, or that we know we will never want to execute in machine code as they are uncommon or they are always going to be slow even in machine code so there is no point trying to optimise them. This is often called an *uncommon trap*, with trap referring to the jump to the interpreter. A key example of an uncommon trap is integer arithmetic overflow. Many dynamic languages have two types of integer, an efficient machine word integer and a much slower heap-allocated arbitrary precision integer. When arithmetic on the machine word integer implementation overflows an arbitrary precision integer is produced instead. This requires code after each arithmetic operation to check if it has overflowed (or alternatively before to check if it will overflow), and code to handle the conversion to arbitrary precision if it has.

Overflow is probably rare in many cases, so more compact machine code can be produced if instead of handling the overflow case in the machine code an uncommon trap is used, the program transfers to the interpreter to handle the overflow.

This is effective on its own, but it becomes even more effective when the program after the arithmetic operation is considered. If the overflow case is compiled into the machine code, then all code after the arithmetic operation will

need to be able to handle either a machine word integer or an arbitrary precision integer. If the overflow is turned into an uncommon trap then we can guarantee that if the machine code is still running then the value must have fit in a machine word integer.

Dynamic deoptimisation can be triggered by the thread which is running the optimised code, in which case it is *local*, or it can be caused by another thread running other code, in which case it is *non-local*. An example of this situation is where one thread redefines a method that has been compiled as an inline cache into machine code being run by another thread. There the redefining thread has to cause another thread to deoptimise.

3.1.6 Inlining

When generating machine code in dynamic optimisation it is possible to replace calls to methods with a copy of the body of the method, in a process known as *inlining*. Inlining is often called *the mother of all optimisations* because it increases the scope over which other optimisations can be applied. This has a significant impact on reducing the cost of high abstraction that is present in many dynamic languages.

Inlining in a language with dynamic and late binding would appear to be problematic, but if an IC is used then the method which has been cached can be inlined. If a PIC is used then multiple methods can be inlined. If methods that have been inlined are redefined then the cache where they were inlined from can be invalidated and the method replaced via dynamic deoptimisation. Inlining can complicate dynamic deoptimisation however, as now a single optimised stack activation can contain an arbitrary number of source-language methods.

3.1.7 Escape Analysis and Allocation Removal

Dynamic languages such as Ruby often work at a high level of abstraction. Data structures such as arrays and maps are used liberally and memory management is automatic so the programmer sees very little evidence in the source code of the cost of the allocation that will be needed. Depending on the language implementation, even fundamental values may be heap-allocated, such as boxed integers or floating point values.

An example of this is Listing 2.11, where we showed a Ruby implementation

of a clamp operation that constructed an array of three elements, sorted it and selected the middle element. Logically, this allocates an initial array and a new sorted array. This can be compared to the C version of the same method in Listing 2.12 which does no allocations.

Allocations are expensive as they access the heap which may utilise cache less efficiently than the program stack would (which will almost certainly already be in cache), they may require synchronisation such as if the heap needs to be expanded, and they create additional work later for the garbage collector. Avoiding allocations to remove this extra level of abstraction is therefore important in dynamic languages. The key techniques to achieve this are *escape analysis* and *allocation removal* [21].

Escape analysis determines if an allocation is ever referenced from outside a compilation unit. Escape is primarily caused by writing the object to another object which is itself escaped. Allocation removal uses the result of escape analysis to determine what objects do not need to be allocated on the heap. The simplest alternative is to allocate objects on the program stack instead with exactly the same layout, but it is also possible to allocate purely in registers if the objects are small and registers are available, and it is also possible to not formally allocate any space anywhere and instead connect producers of values which go into an allocated object directly with their consumers which would have read from the object, in the program's IR. Allocation removal may not be ideal in all cases even if an object does not escape, such as if an object would consume a large amount of program stack space (program stacks are difficult to reallocate if space is exhausted).

Even the HotSpot C2 compiler has only supported escape analysis only since version 1.6, so while the technique is well understood it isn't always widely applied.

3.2 Applied Techniques

3.2.1 Self-Optimising AST Interpreters

One of the two key technologies applied in this thesis is the *Truffle* language implementation framework for *self-optimising AST interpreters* [115].

Truffle began with the observation that most language implementations begin

with an AST interpreter and then go through a long process of slowly adding the more sophisticated runtime techniques described above as they gain users and those users demand better performance.

When a Truffle AST node's compute action method is run, as well as performing the compute action the node can replace itself with a specialised version based on the data it received. The simplest example of a specialisation is one based on the types of values received from child nodes. An add operation in many programming languages is polymorphic and may be able to add together integers, floating point values, strings and so on. The add node in a conventional AST interpreter will therefore also be polymorphic. In Truffle there are multiple implementations of the add node, each designed to handle specific types. When a Truffle AST is run the uninitialised add node will replace itself the first time it runs with a new node specialised for that types that it has previously seen. If different types are seen the next time that node is executed, it may specialise again. In order to prevent continual re-specialisation, a lattice of types may be used so that specialisations always move towards a megamorphic node which handles any types. This megamorphic node is effectively the same as the node that would be used in a conventional AST interpreter.

Figure 3.1 shows the Truffle model of specialisation. Initially we will only focus on the first two of the three stages shown. An AST begins with all nodes in uninitialised state, shown with the labelling of nodes as U. As the program runs and nodes see values produced by child nodes, nodes can *specialise* by replacing themselves with versions that are suited to the type of those values. A node that sees two integers, shown by I, produced by its children may rewrite as a version specialised just for integers. So that these specialisations converge, the types involved form a lattice and any specialisations usually move downwards towards a completely general case which can handle any types. The lattice in this diagram shows primitive integer I and double precision floating point D types, a string S type and a generic G type. A generic node can handle any types and so does not specialise further.

AST specialisation can be seen as similar to, but more general than bytecode quickening [19]. One differentiator is that nodes do not have limited space to expand into, and a node can specialise itself to a new sub-tree of nodes, rather than replacing nodes one-for-one. An example of this is the dispatch chains that will be shown in Chapter 5.

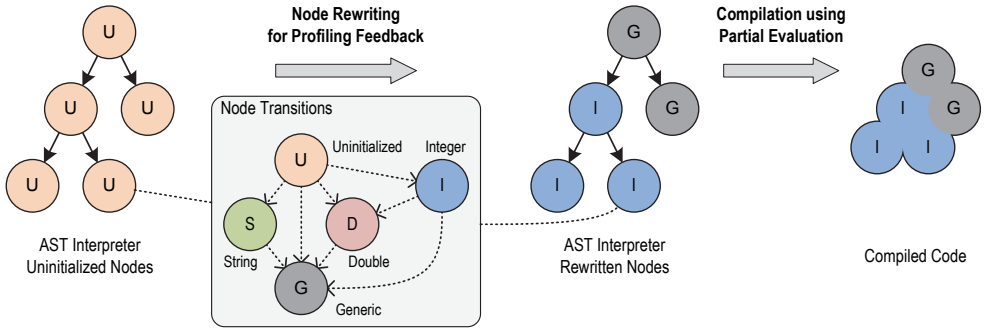


Figure 3.1: Self-optimising ASTs in Truffle [114]

Self-optimising AST interpreters have proved to be a simple primitive which can be used to represent many of the essential techniques of a high performance language implementation. For example, ICs and PICs can be represented as chains of nodes, with each containing the guard and a method to call.

Truffle is implemented in the Java programming language by a team of researchers at Oracle Labs. Truffle node classes can be written by hand in Java, or a DSL based on Java annotations can be used to automatically create nodes based on multiple execute methods, with the types of their arguments declaring the types that the node for that execute method will accept [56]. The Truffle DSL has many sophisticated features which are not discussed further in this thesis, such as creating a chain of nodes to handle multiple types and avoid moving to a megamorphic node, facilities to declaratively create PICs and more.

3.2.2 Partial Evaluation

Truffle improves the performance of AST interpreters, but on its own it does not help language implementations achieve the kind of performance that is possible with dynamic optimisation. To improve performance further, Truffle is combined with a new dynamic compiler for the JVM, *Graal* [114].

Graal began as part of the Maxine project, which was a *meta-circular* reimplementation of the JVM in Java [111]. A meta-circular implementation of a language is one that is itself written in that language. In the case of Maxine, Java was used to implement the JVM. Maxine's optimising compiler, C1X, began as a port the HotSpot C1 compiler into Java. C1X was put back into HotSpot as C1X4HotSpot, and later became Graal with significant new optimisations that make it comparable in performance to HotSpot's peak performance compiler C2.

Graal now uses a sea-of-nodes [22] IR, similar to C2, that is designed to be extensible [26] so that projects such as Truffle can be integrated easily. Today Graal is at least competitive with the performance of C2 [98].

Graal is not writen in Java just for the sake of academic exercise in meta-circular implementation. If the compiler is written in Java then it can expose a Java API to the running program. Truffle's Graal-specific backend uses this API to directly control the compiler as a service, and use it to compile Truffle ASTs after they have been specialised.

The final transition of Figure 3.1 shows how Truffle's Graal backend takes all of the execute methods from the nodes in an AST, inlines them all and all methods they call into a single compilation unit, applies optimisations, and produces a single piece of machine code for the whole AST. The technique that Graal uses is called *partial evaluation* [93, 103]. This means that the compiler does not just compile the program, it will also execute as much of it as it can, given the information it knows at compile time. Truffle treats the AST data structure as information known at compile time, resulting in a residual program that runs the program that the AST described. This transformation is known as the first Futamura projection [34].

Graal has very good support for speculative optimisations [27], and is able to deoptimise from the compiled, partially evaluated representation of an AST back to the unoptimised AST intepreter[1]. Truffle's Graal backend will automatically deoptimise if any nodes re-specialise, and it also provides two mechanisms to explicitly control deoptimisation. Threads can explicitly deoptimise using `transferToInterpreter()`, they can deoptimise and invalidate the current machine code using `transferToInterpreterAndInvalidate()`, and a third option designed to support non-local cache invalidation is an `Assumption` object which is referenced in compiled code and has a method to `invalidate()` that transfers and invalidates the locations in compiled code which referenced that object.

Figure 3.2 shows how Truffle's Graal backend uses deoptimisation to be able to continue to specialise ASTs even after they have been compiled. Control transfers from the compiled machine code back to the AST. Additional specialisations may be triggered by whatever change in state it was that caused the deoptimisation

[1] Note that the Truffle interpreter may itself be compiled by the conventional JVM dynamic compiler, but we refer to this state as the unoptimised interpreter for simplicity.

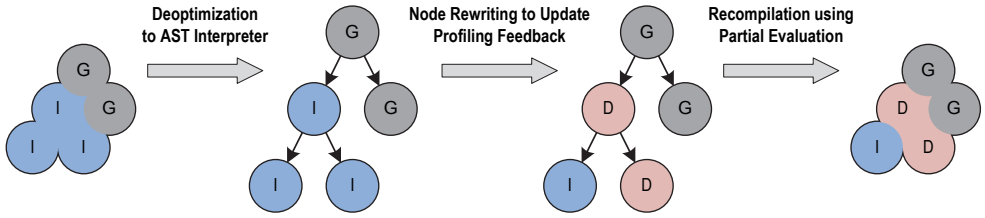


Figure 3.2: Graal and Truffle deoptimising from optimised machine code back to the AST, and re-optimising [114]

in the first place. After allowing for stabilisation, optimisation may again be triggered and new machine code is produced.

3.2.3 Partial Escape Analysis

In conventional escape analysis, as soon as it is determined that an object may escape at some point, the object will be considered escaped in all cases. Graal supports a more sophisticated *partial escape analysis*, which will consider on which branches an object escapes, and will allocate the object only as control flow reaches a point where the object will escape [96, 101]. This allocation is called *reification*, *re-materialisation* or *de-virtualisation*.

In a method with an object that is logically allocated ahead or two possible branching control flow paths, on one of which the object escapes, and another where it does not, conventional escape analysis would reify the object at the allocation point, as the object would be considered escaped due to the one possible control path where it escapes. Partial escape analysis would reify the object only at the point where the control flow edge is taken which means that the object will actually escape.

For Truffle's Graal backend, the sophisticated partial escape analysis phase is not just another optimisation phase. Truffle's represents runtime structures such as stack activation frames as normal Java objects, and relies on the partial escape analysis phase to successfully remove their allocation.

3.2.4 Splitting and Inlining

Truffle's AST model allows for conceptually simple inlining. To inline one method (so one AST) into another, a node that would have made a call is replaced with a node that references the called AST. Unlike in compilers which perform inlining

at a lower level IR, no special work is required to modify argument handling in the inlined method. Inlined methods can just allocate their own new stack activation object as normal and partial escape analysis is again relied upon to collapse multiple stack activations into one.

Splitting is a related technique, where an inlined AST is not just referenced from the call site, but is also copied so that the call site has a separate copy of the method, which can specialise independently of any other.

3.2.5 Other Dynamic Language Optimisation Techniques

Truffle's Graal backend is a method-at-a-time JIT, with additional support for *on-stack replacement* (OSR) of a running method. OSR allows a method that has entered a long-running loop to be transfer into compiled code while it is still active on the stack.

Examples of more conventional method JITs include Google's V8 engine and Mozilla's JägerMonkey and IonMonkey JavaScript implementations, the JRuby and Rubinius implementations, as well as most JVM and CLR implementations.

The key alternative to method-at-a-time JITs is *tracing* and *meta-tracing* compilers [58, 16, 14, 18]. A tracing JIT compiles loops, rather than methods, and any code that is dynamically part of the loop is liable to be included in the compilation unit which means that inlining comes naturally. The term tracing refers to the way that the behaviour of the program is traced to produce a sequence of instructions that were on the critical path and should be compiled. Meta-tracing means that the instructions being traced are not those of the application itself, but those of an interpreter for the application. PyPy's RPython language implementation system [16], LuaJit [83], and Mozilla's TraceMonkey JavaScript implementation [36] are three key examples of tracing JITs.

Less common approaches for optimising dynamic languages include *block-at-a-time* or *tracelet*-based JITs such as Facebook's HipHop VM for PHP [8] and the Higgs JavaScript implementation [20].

3.2.6 Additions to Truffle

Truffle and Graal are existing research projects [115, 114]. The additions for debugging and tracing in Chapter 7 are now part of the public Truffle API. In the course of this research, some additions were also made to the `Assumption`

class, described in depth in Section 7.5.

3.3 Implementations of Ruby

3.3.1 Matz's Ruby Interpreter

As with similar languages such as Python, Ruby has a single canonical implementation, with the behaviour of this implementation defining the behaviour of the Ruby language. In the case of Ruby this is *Matz's Ruby Interpreter* [116], named for the creator of Ruby, Yukihiro Matsumoto. In some cases it is also called *CRuby*.

The original implementation of MRI was a simple (not self-optimising) AST interpreter implemented in C with a global method cache. This was the implementation used through to version 1.8 of Ruby, which was the period where Ruby became popular with the Rails framework and was in use until around 2007. This version of Ruby was standardised as ISO/IEC 30170:2012 [57].

After Ruby became popular with the Rails framework there was a desire to improve the extremely limited performance. Around 2005 MRI was forked by Koichi Sasada and the execution model was rewritten to use a bytecode format and interpreter, rather than directly interpreting the AST, and to use inline method caches [90]. This fork, known as YARV, was merged back into MRI for the 1.9 version of Ruby and increased performance by around 2× (see Section 3.3.6).

3.3.2 Rubinius

Rubinius [86] was started as a from-scratch implementation of Ruby, using the Blue Book techniques that were applied in the implementation of Smalltalk [38] (some of these techniques were discussed in Section 3.1) in order to try to achieve higher performance. Rubinius supports the majority of the Ruby language and libraries and are able to run significant applications in production.

The Rubinius VM is implemented in C++, but a substantial portion of the Ruby-specific functionality such as the parser, bytecode compiler and core library is implemented in Ruby. The first stage of execution in Rubinius is a bytecode interpreter similar to that in MRI. In the next tier Rubinius uses LLVM [65] to implement JIT compiler backend. The Rubinius compiler is not much more than

a template compiler, emitting a pattern of LLVM IR for each bytecode instruction and relying on the optimisation phases provided by LLVM.

Rubinius has a simple form of dynamic deoptimisation in that it can transfer from compiled code back to the interpreter and it can invalidate machine code, but Rubinius rarely uses uncommon traps and often uses fallback code in machine code rather than deoptimising on case that the machine code was not designed for.

In practice, the performance of Rubinius on all benchmarks we tried was not much better than that of MRI (see Section 3.3.6).

3.3.3 JRuby

JRuby [78] is an implementation of Ruby written in Java and running on the JVM. Like many other implementations of Ruby it began as a simple AST interpreter and has adopted more sophisticated techniques over time. JRuby 1.7 compiles ASTs to JVM bytecode [41], without an intermediate bytecode as in Rubinius, using a template compiler that visits the AST to emit a pattern of byecode for each node. As with Rubinius and LLVM, JRuby relies on the JVM to apply almost all optimisations beyond this point.

Like, Rubinius, JRuby also supports the majority of the Ruby language and libraries and is able to run significant applications in production.

Invoke Dynamic

In an attempt to improve on the performance of dynamic languages on the JVM in general, the Da Vinci Machine project added a new instruction to the JVM, `invokedynamic`, and associated runtime infrastructure. This new instruction allows application-defined call site semantics [87, 89].

JRuby was a key-motivator and an early adopter of `invokedynamic`. Until the development of the new Nashorn JavaScript implementation, and implementation of lambdas on Java 8, JRuby was easily the most sophisticated and widely deployed application of `invokedynamic`.

After several years of refinement, performance of `invokedynamic` for JRuby is still perhaps not as good as was hoped. For the benchmarks that were evaluated, JRuby with `invokedynamic` was often no faster than without, sometimes worse, and never more than around twice as fast in exceptional cases. Overall

it is only 13% +-5% better than without, for the benchmarks that we tried (see Section 3.3.6).

JRuby IR

In both Rubinius and JRuby, the only optimisation phases are those provided by the underlying system which is LLVM or the JVM. The JRuby project is attempting to tackle this by using a high-level IR that represents Ruby code more directly than JVM does, and can be used to manipulate the program to apply optimisations such as constant folding.

At the time of writing, the JRuby IR does not demonstrate an overall speedup compared to the old AST template compiler (see Chapter 4), but this is because the foundation for IR has only just been finished and significant optimisations have not been added yet.

3.3.4 Topaz

Topaz [37] is an implementation of Ruby written in Python using the RPython language implementation system and its meta-tracing JIT [16]. Topaz uses a bytecode format and the RPython implementation is an interpreter for this bytecode, which is meta-traced for produce a JIT.

Unlike MRI, Rubinius and JRuby, Topaz is at an early stage in development. It supports around 44% of the Ruby language and around 46% of the core lirbary (see Section 4.4 for methodology), which is not enough to run much beyond the small benchmarks evaluated here.

3.3.5 JRuby+Truffle

JRuby+Truffle is an implementation of Ruby using the Truffle language implementation framework and the Graal dynamic compiler. Originally a standalone research implementation and known as RubyTruffle, JRuby+Truffle was later open-sourced and merged into the exist JRuby project as an optional backend.

JRuby+Truffle supports around 93% of the Ruby language and around 87% of the core library. This is enough to run simple web frameworks such as Sinatra [75], but not yet Rails.

JRuby+Truffle is conceptually at least one level simpler than other contemporary implementations of Ruby, as it is still just an AST interpreter, although

a very sophisticated one. It does not have a bytecode format and has no manual code or IR generation phase. All the stages beyond the AST interpretation and specialisation are handled automatically by Truffle and the Graal compiler.

3.3.6 General Performance Evaluation

Before describing the research contributions of this thesis, it is important to consider a very broad, general overview of the performance of the different implementations of Ruby. Some of the techniques we describe such as in Chapter 5 aim to achieve zero overhead for using a particular language feature. However, this is not useful if the baseline performance is poor.

Methodology for all of the evaluation in this thesis is discussed in Chapter 4.

Figure 3.3 shows speedup compared to MRI 2.2.2 for the released version of JRuby with `invokedynamic`, Rubinius (abbreviated Rbx), Topaz, the development version of JRuby with IR and also `invokedynamic`, and JRuby+Truffle running with the Graal compiler. Performance for individual benchmarks varies considerably; more precise results will be presented in specific chapters and this figure is only intended to show that the performance of JRuby+Truffle is at the very least extremely competitive.

For this set of benchmarks, neither Rubinius nor JRuby manage to achieve a meaningful speedup compared to MRI, but JRuby+Truffle achieves and order of magnitude increase in performance over MRI.

Figure 3.4 shows the same results, but without JRuby+Truffle so that the other implementations may be more easily compared. Here the performance lead that Topaz has over the other implementations is clear.

Figure 3.5 shows performance on historical versions of MRI. An increase in performance is shown between versions 1.8 and 1.9 when YARV is merged. The subsequent increase in performance between 1.9 and 2.0, and 2.0 and 2.1, was probably caused by improved GC and other smaller optimisations. This shows that the performance of the runtime system is probably as important as improving the execution model.

Figure 3.6 shows the impact of `invokedynamic` on the performance of JRuby for the benchmarks we evaluated.

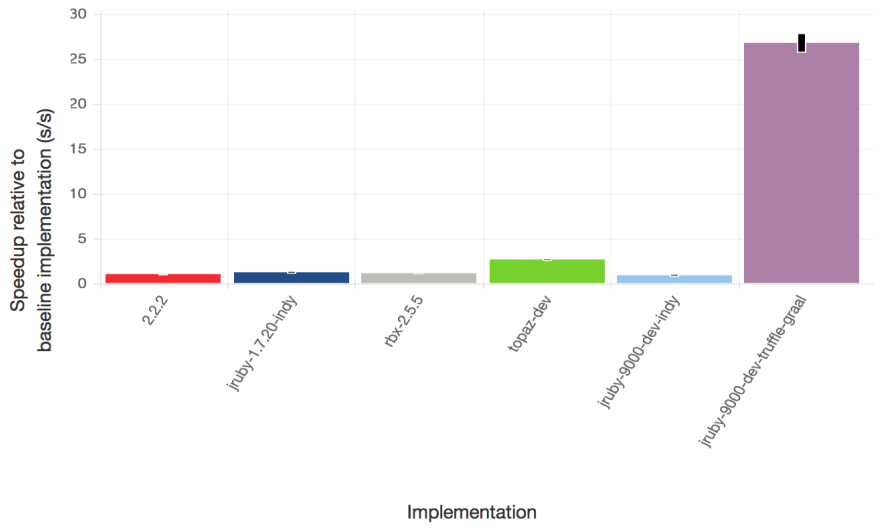


Figure 3.3: Summary performance of different Ruby implementations on all of our benchmarks

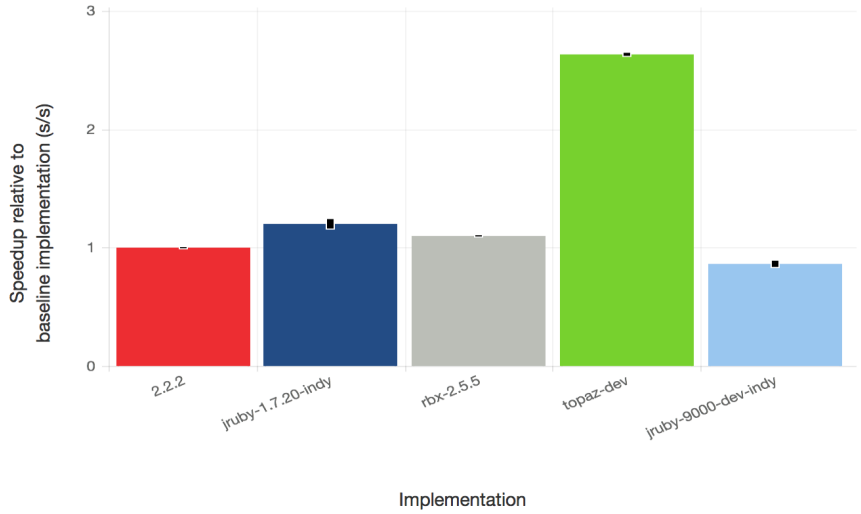


Figure 3.4: Summary performance of different Ruby implementations on all of our benchmarks, excluding JRuby+Truffle

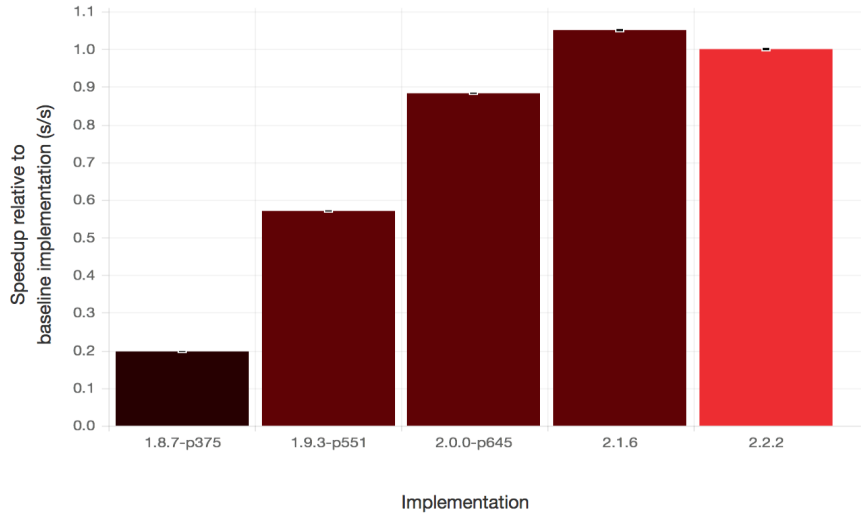


Figure 3.5: Summary performance of historical versions of MRI on all of our benchmarks

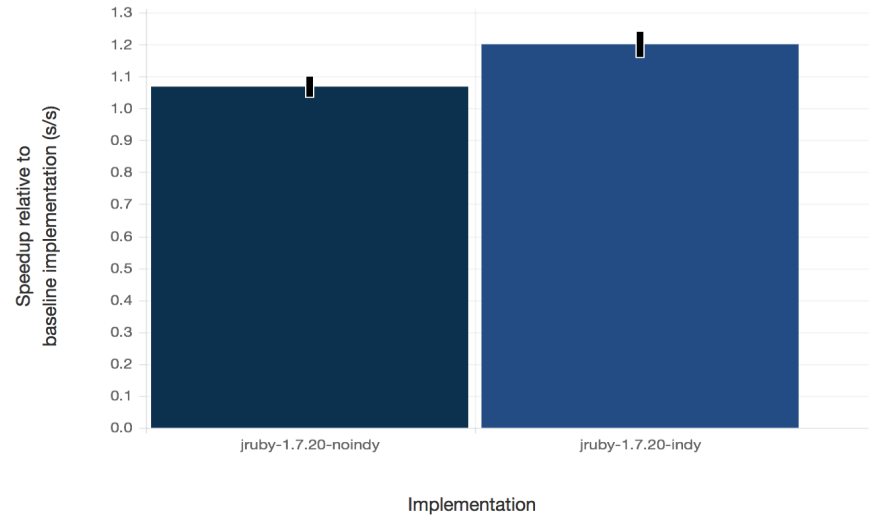


Figure 3.6: Summary performance of JRuby with `invokedynamic`

3.3.7 Other Implementations of Ruby

MRI, Rubinius and JRuby are the only implementations of Ruby that are sufficiently complete and up-to-date with the Ruby language for there to be significant deployment in production. Compared to other similar languages such as Python, there have been a large number of Ruby implementations over the years, although most of them are now discontinued and none of them achieved performance even comparable to modern JRuby or Rubinius.

MagLev [30] is an implementation of Ruby built on top of the GemStone Smalltalk VM which provides a JIT. The unique feature of MagLev is that it provides a distributed shared cache and a persistent object store with ACID transactions. We were not able to benchmark MagLev due to licensing concerns, but performance is known to be comparable to MRI 1.9, which when it was originally released (in the 1.8 era) was a significant result [7].

Ruby Enterprise Edition was forked from MRI 1.8, when it was still using a simple AST interpreter and performance was extremely limited. REE modified the garbage collector to interact better with copy-on-write when the forking model of concurrency is used, and added an alternate implementation of `malloc`. Claimed benefits at the time were reduced memory usage and increased performance, although these were easily surpassed by the later YARV work that was merged into MRI.

MacRuby and RubyMotion are related implementations of Ruby that use the integrate into the Apple development ecosystem such as supporting the Cocoa API on the Mac and UIKit API on iOS. MacRuby included a JIT compiler using an LLVM backend, like Rubinius. RubyMotion also supports compilation to JVM bytecode, and reportedly a Windows target is in development. RubyMotion is a commercial product so it was not possible for us to evaluate the performance or compatibility it achieves. The RubyMotion documentation says that it only supports 'a dialect of Ruby that is statically compiled' and although we cannot verify, we think it is unlikely that it either supports metaprogramming that we are interested in, or that it implements it with the high performance that our benchmarks require.

mruby is a very simple implementation of Ruby designed for embedded environments. It does not aim to be compatible with standard Ruby.

IronRuby was an implementation of Ruby on Microsoft's Dynamic Language Runtime effort to improve dynamic language support on the CLR [3].

XRuby and InfraRuby were both statically compiled implementations of Ruby targeting JVM bytecode. InfraRuby can also be statically typed using non-standard type annotations and type inference.

Ruboto is JRuby packaged for the Dalvik VM and the Android platform.

HotRuby, RubyJS and Opal are all implementations of Ruby that are designed to work on a JavaScript platform. HotRuby is a JavaScript interpreter for the bytecode generated by YARV. RubyJS is just an implementation of the core library to allow people to program in JavaScript but use a familiar library. This may be designed to be used with the CoffeeScript language, which is a language with Ruby-like syntax that is translated to JavaScript. Opal is a Ruby to JavaScript translator. Of these three, only Opal is maintained and attempts to implement a significant portion of the Ruby language and core library.

Crystal is a statically typed and statically compiled language that has has a similar look and feel to Ruby, but does not support any forms of metaprogramming. It uses the LLVM framework for static compilation.

We did not evaluate against these implementations either because they are unmaintained, they do not support modern Ruby, they do not support enough of the standard Ruby or core library to run our benchmarks, their performance is known to be extremely limited, or they do not implement features which are relevant to our evaluation such as metaprogramming and debugging.

3.4 Parallelism and Concurrency

Most implementations of Ruby support concurrency, with the `Thread` and `Fiber` objects allowing for pre-emptive and cooperative concurrency. This is useful in practice for IO operations, which may block one thread but allow another to continue until the result is available. Only Rubinius and JRuby support parallelism

for the `Thread` object, which means actually executing more than one thread at the same time on multiple cores. Other implementations of Ruby have a global interpreter lock which can only be held by one thread at a time, to actively prevent parallelism. JRuby+Truffle currently also has a global interpreter lock. This is not an inherent limitation of the Truffle and Graal approaches, but removal of the lock is still a fairly significant project and is not part of this thesis.

3.5 Summary

We have described existing techniques for implementing dynamic programming languages. Many of these techniques are well understood but they aren't always applied in implementations of Ruby due to the complexity of implementation. A system such as dynamic optimisation is a great deal of work, and previous efforts to re-use that work from systems such as LLVM or the JVM have not shown a huge increase in performance in Rubinius or JRuby for the benchmarks tested.

We think that this could be due to the existing techniques not working well for the characteristics that were identified in Chapter 2. Novel techniques are needed to optimise these Ruby language features, and that will be the contribution of the following chapters.

In this chapter we also introduced JRuby+Truffle and the other implementations of Ruby that we will be comparing against, and stated why we are not comparing against legacy implementations of Ruby that are unmaintained or do not support the features we are evaluating.

Chapter 4

Evaluation Methodology

In this chapter we describe the key goal for the majority of work in this thesis. We begin by describing the context in which Ruby is used and use this to inform a philosophy for our evaluation and identify the goal that we are working towards. We then set a metric that will be used to evaluate if that goal is being reached, and describe how we measure, summarise, compare and present that metric. We also identify non-goals which fall outside of the scope of the work in this thesis.

In Section 3.3.5 we already provided an overview of the performance of the system implemented for this thesis, JRuby+Truffle. This chapter gives the methodology used for that overview, and for the evaluation sections in the following chapters.

4.1 Evaluation Philosophy and Goals

We begin with an observation about the environments in which Ruby is used. Ruby is generally used in a server environment, when long-running processes handle a non-terminating stream of requests, each of which must be processed and a response sent.

The customer or other end-user accessing a web server cares only about the time taken for the request response to be returned. Reducing response time by increasing the compute power in a server is difficult, due to the sequential nature of most Ruby programs and implementations, and the slowing gains in sequential single core performance of processors. This means that adding additional cores will not reduce the response time for a single customer. Increasing the memory capacity of a server is more easy as the limit on the quantity of memory that

can be attached to a server is very high and a single sequential process can take advantage of all memory attached to a system. From this we identify that the goal is to minimise the time taken to produce a response to a request, and that we want to do that by reducing the compute resource used to handle a request, possibly at the expense of using additional memory.

As Ruby programs generally run indefinitely, handling a stream of requests until the service is upgraded to a new version, is no longer needed, or maybe some other conditions such as migrating a service between servers, we identify that the goal is more specifically to reduce the average response time in a long-running process. The time needed to fulfil an initial sub-set of requests is not as important, which means we may want to minimise the response time of the majority of requests, possibly at the expense of the response time for initial requests.

Our goal therefore is to minimise the time needed to perform some computation iteration in a Ruby program, as part of a long running process, allowing for an initial set of iterations that can require more time.

4.1.1 Peak Temporal Performance

The metric that we use to evaluate performance toward this goal is *peak temporal performance*.

By *temporal* we mean the time taken to complete an iteration of some computation, such as a request made to a service. We are concerned with real elapsed wall-clock time, as that is what the individual user or customer perceives waiting for a response. The customer does not differentiate between time spent in kernel processes or user processes, or time spent in garbage collector or compilation, so we 'bill' all time consumed by any part of the system to the benchmark.

By *peak* we mean the period of the process after it has had sufficient time to reach its best performance. In practical terms and with reference to Chapter 3, this means when the system has had a chance to gather profiling information, to populate caches and to perform dynamic optimisation.

It would seem that it should be simple to determine the time when peak performance starts. If we are measuring iteration sample times then maybe we can observe when the times suddenly drop after dynamic optimisation. Or if we can get feedback from the dynamic compiler maybe we can be notified when dynamic optimisation is complete.

The problem with looking for a drop in iteration times is that although a drop may have been seen, there is no way to know if there will be a further drop in the future. With extremely sophisticated interaction with multiple systems such as GC, it is also possible that runtime times could go up after a period of time.

The problem with using feedback from the dynamic compiler is that different parts of the program will be compiled at different times and there is in practice no big bang when compilation is complete. We informally experimented with using this technique and found that when running benchmarks for many minutes methods would occasionally still be compiled, such as those involved in the benchmark harness, or methods only called once per iteration. Dynamic compilers like Graal are also entitled to compile whatever methods they want at any point, or even to recompile a method based on new information.

In general, determining when a system has warmed up, or even providing a rigorous definition of the term, is an open research problem that we are not able to provide a solution to here.

Where we use the term *performance* without qualification, we mean peak temporal performance. A related term to peak performance is *steady state*. An implementation is running in a steady state if performance is not currently changing, or more strongly if it will not change in the future.

4.1.2 Non-Goals

There are many other legitimate goals for the implementation of a dynamic programming language which fall outside of the scope of this thesis.

Start-Up Time

Start-up time is the time taken before the first line of the benchmark is executed. The starting point for this measurement could be the point before the parent process calls `exec`, or it could be in the first line of the `main` method of the implementation.

The start-up time metric measures how fast the system can load and parse source code, how fast any standard library code can be loaded, how long the runtime takes to setup threads for services such as garbage collection, and so on. Start-up performance has a big impact on certain use cases such as programs written to be used as command-line utilities or as part of shell scripts. This is

the case of Ruby for some limited applications, and JRuby is often criticised for its poor start-up performance.

We don't consider start-up in this thesis as it does not impact on the use-case of Ruby that we are interested in and that we have used to set our goal[1].

Cold Temporal Performance

We do not consider cold temporal performance, which would mean the time taken to perform a computation iteration for the first time in a process. In the case of JRuby+Truffle and similar implementations, the cold performance is the phase in which the interpreter is running and in which ASTs are being specialised and profiling information gathered.

JRuby+Truffle deliberately accepts a trade-off of time to warm-up in return for higher peak-performance. As a concrete example, a pair of branches which are chosen between based on a Boolean condition in JRuby+Truffle are ordered in the compiled machine code based on the probability of each branch being taken. When they have no other information, either because the branch has never been seen or the branch prediction cache is full, common processors will assume that a backward jump is taken an a forward jump is not taken. We use this knowledge to put higher probability branch behind a backward jump, and the lower probability branch behind a forward jump, regardless of the original structure of the branches in the user's source code. In order to know the probability of the two branches, the number of times each branch is taken must be measured and stored. This adds overhead to the cold phase where the interpreter is running, as an expense to improve the performance of the warm phase when the optimised compiled code is running.

If the time taken to respond to every request is critical and initial slow responses cannot be tolerated, it is possible to send an initial stream of synthetic requests designed to warm the system up to peak performance before the first live request is received.

[1]For JRuby+Truffle the start-up time problem is being solved using the SubstrateVM, an ahead of time static compiler and that builds a single statically linked binary version of JRuby+Truffle, Truffle and Graal with no dependency on the JVM. This will give us fast start-up as the whole native image, initialised and ready to start, can just be loaded via `mmap` and started. It should also give us the same peak performance of JRuby+Truffle running on a JVM, as the same dynamic compiler is available.

Time to Warm-up

If we know that there is a phase where performance may be lower followed by the phase of peak performance then at some point there is a transition. This transition between phases is called the point where the program *warms-up*. The time taken to finish the initial low performance phase and to finish the transition to high performance is called the *time to warm-up*.

Environments where warm-up time is more important include command line utilities used by developers such as a compilers, but these are less common applications for Ruby and for our stated goals time to warm-up is not important as long as it is reasonable.

As with cold temporal performance, time to warm-up is not evaluated throughout this thesis. As was discussed in Subsection 4.1.1, a technique for determining the atomic point in time where a system has warmed-up is not known, and it is not clear how to quantify and visualise such a metric, especially in summary across a range of benchmarks.

Memory Consumption

We already stated that memory consumption is out of scope for this thesis, because in the environments that Ruby is run it is easier to scale memory than processor power, and because customers will typically only be able to observe response time. As well as being out of scope, memory consumption is an extremely problematic metric for a dynamic language implementation.

A key problem is the behaviour of the garbage collector. A tracing garbage collector, that periodically recovers memory that is not reachable by the program, operates in a cycle of increasing memory use and then sudden reclamation. This means that the volume of memory allocated at any point depends on where in the cycle we are, and the logic used to determine how far to let allocated memory expand before memory is reclaimed.

With a tracing garbage collector it is not clear how to answer the simple question 'how much memory is this program using?' We can determine how large the heap is at a given point, but this will include objects which could be collected if we wanted to. We can determine how large the heap is after a collection, and so when it contains only objects which are reachable, but this is not the space actually needed. A garbage collector will use the memory that you make available to it, so that an application configured to have a larger heap size will. Further,

the application may create caches that are only cleared when there is memory pressure. This makes us consider whether optimising for low memory usage is even a desirable goal. If we have memory available in the system, why not use it?

We end up with a problem with two degrees of freedom. We can discuss the performance of a system only with a given volume of memory available, and we can discuss the memory consumption of a system only with reference to the performance that this constraint produces.

We do evaluate memory consumption in that we include the time needed for garbage collection in the measured time for each iteration. Optimisations that we have made in JRuby+Truffle such as allocation removal of temporary data structures probably reduce runtime because they reduce the load on the garbage collector–and in fact they would also reduce the load on a manual allocator if we were using one. Beyond that, memory consumption is not evaluated throughout this thesis.

4.2 Benchmarking Language Implementations

Evaluating the performance of dynamic language implementations through benchmarking is highly problematic, and there exists no single universally accepted methodology. Here we will describe a range of techniques and discuss whether they are appropriate in evaluating out stated goal.

4.2.1 Basic Techniques

The naive technique for benchmarking a language implementation is to run a program from start to finish and report the wall-clock time. This can be achieved very simply with the `time` command-line utility. This metric includes start-up and cold performance, and may include time to warm-up and then peak performance if the program runs for long enough. Therefore it conflates multiple metrics.

A more sophisticated technique is to run the benchmark in a loop, measuring the time for each iteration. This excludes start-up time because only program code is measured. It does not hide cold performance, as the first iteration of the benchmark will be cold, but it does allow the cold performance to be excluded by discarding the first iteration. It also does not hide time to warm-up, but again

the iterations of the benchmark while it was warming-up can be discarded. As we said in Subsection 4.1.1, determining the atomic point in time when an implementation has warmed up is extremely problematic. A simple solution is to allow the implementation an extremely generous time to warm-up before iterations are sampled. It is also possible to use a simple metric such as the range of the samples from a window of iterations to check that this has been successful.

4.2.2 Advanced Techniques

Automatic Iteration Scaling

A question that has yet to be solved is how long to make each iteration of the benchmark. System timers have finite precision and resolution so timing a computation that is very small is not likely to be sound. Most benchmarks will have some parameter that we can vary to set the time for each iteration. For example the Mandelbrot benchmark has a parameter for the size of the image to generate. Smaller benchmarks that do not have such a parameter can be made to run for a certain period of time by putting them in a loop of some constant length. A problem that this can cause with optimisation is discussed below.

In JRuby+Truffle we determined parameters for the benchmarks which made them generally take around $2s$ on JRuby. This means that the slower implementations, such as MRI don't take over a minute for each iteration, and the faster implementations such as JRuby+Truffle are not so fast that precision in reported iteration times is not lost.

Another technique is to spend some time determining how fast an implementation is running a benchmark, and then set a number of iterations so that each iteration takes some fixed period time, such as a few seconds. This is the approach used by the popular Ruby benchmark utility `benchmark/ips`.

We discarded this approach, because it is only practical where the size of the benchmark is varied by a number of iterations of an inner loop and not by a more complex parameter such as the Mandelbrot size for example, which does not scale the problem size linearly. It also effectively means that each implementation is running a benchmark with different parameters, and with timing calls that are different intervals.

Layers of Iteration

We are discussing running iterations of a benchmark in a loop, measuring the time taken for each iteration. However, we could loop iterations at a level above this, by also running the implementation process multiple times. This We can keep examining further layers of the system, and could experiment with compiling the implementation multiple times with variation such as randomised link ordering.

We did not experiment with multiple layers of iteration for reasons of simplicity and limited compute resources. Researchers who are trying multiple layers of iteration have seen time needed for benchmarking increase to levels that would be unsustainable for this project. Informally, our continuous integration system that benchmarked every commit we made when developing the system showed us that our results are stable across multiple invocations of the implementation process.

Replay Compilation

One option to improve the repeatability of an experiment is to record compilation decisions taken by the dynamic compiler, and to perform them exactly the same each time the benchmark is run.

We discarded this option because as because we have already ruled-out multiple layers of iteration it is not relevant. Also, replay compilation reduces the normal flexibility afforded to the dynamic compiler.

Isolating Garbage Collection

Some researchers also choose to control for garbage collection. One option is to disable the garbage collector for the entire run of the benchmark. This is not practical for benchmarks that run for anything beyond a trivial length of time and that allocate objects as they may very quickly run out of heap size, and it is also possible that it will reduce benchmark performance, as an increasing heap size may mean that locality suffers.

Another option is to perform a full garbage collection before each benchmark iteration, in an effort to give each iteration a ‘clean slate’ and to reduce interdependence between iterations. However some garbage collectors may not honour requests for a collection from the user application, and if a full collection were performed it may evict benchmark code and data from caches and so actually

reduce performance of the next iteration.

A third option is to calculate the time spent performance garbage collection during a benchmark iteration, and to subtract that time from the sampled time. This produces an even more artificial sample than the other two options–one that never existed at all.

We disregarded all of these options as we consider garbage collection to be part of the routine cost of 'doing business' in a garbage collected language, which should be attributed to the benchmark.

Preventing Over-Optimisation

One of the key jobs of an optimising compiler for a dynamic language is to remove unnecessary work, and we want to measure how well it does that. However the risk is that the benchmark also becomes unnecessary from the point of view of the compiler, and it is removed partially or entirely. Another risk is that the work remains, but is constant-folded into the result value during compilation.

It is debatable how much of this we should allow. We want to exercise how well implementations are able to remove redundant work, so perhaps we should not try to prevent this optimisation. If an implementation is able to entirely remove a benchmark then perhaps that is what should be measured.

JRuby+Truffle is sophisticated enough to complicate the problem even further, as it has *value profiling* which looks at values at many points in the dataflow graph and will turn an edge that has only ever seen one value into a guard comparing against that value and then a constant. Even if input to a benchmark were read from a file, as long as it didn't change JRuby+Truffle could still constant fold it.

Solutions to this include techniques such as those employed by the Java Microbenchmarking Harness, which has an API for *black holes*–methods that use internal knowledge of the implementation of the JVM to create a dataflow sink that will not be removed[2]. However there is no such API for Ruby and we would not want to add one specifically for JRuby+Truffle without implementing it to the same standard in other optimising implementations as JRuby and Rubinius.

In our benchmarks we decided not to make any special effort to avoid over-optimisation, as we could see no way to avoid it entirely. We have explained why

[2]Note that the Java Microbenchmarking Harness is a state-of-the-art tool, but still requires time to warm-up to be manually specified.

we do not know of a total solution, and any partial solution would be hard to justify–why prevent this part of the benchmark from being optimised but not another part.

Dynamic Frequency Scaling

Some systems will vary the clock frequency to balance power consumption and heat output with the demands of the current workload. Dynamic voltage scaling is a related technique. The risk is that some implementations and iterations will run with frequency at one level and another implementation or iteration with another level.

It could be argued that frequency scaling is something that should be 'billed' to the implementation as a side effect of running a program, similar to garbage collection or virtual memory paging. If the process being benchmarked causes the kernel to decide to scale the processor frequency then it is arguable that it is a factor which should be considered as any other factor is.

In practice, we believe this is a bit extreme and does not help us produce data that can be related between implementations.

The systems on which we benchmarked had dynamic frequency scaling disabled by setting them to the *performance* power management configuration.

4.3 Statistical Techniques

4.3.1 Basic Statistical Techniques

Aggregating Samples

There will likely be variation between sample times of multiple iterations of a given benchmark. We have already established that the performance of implementations will change over time, but even if we are confident they have entered a phase of steady-state we would expect to see variance caused by implementation background processes such as compilation and garbage collection, background system processes and possibly clock inaccuracy.

It is important to note that sequential iterations of a benchmark are absolutely not independent samples. As one iteration runs it modifies the state of the heap and other runtime data structures, processor caches, kernel data structures and more.

We use the simple arithmetic mean average to aggregate multiple samples into a single scalar value. The mean time $\bar{T}$ for n times samples T_1 to T_n is:

$$\bar{T} = \frac{1}{n} \sum_{i=1}^{n} T_i$$

Scoring and Speedup

An absolute mean time per iteration is not a metric that can be compared across benchmarks or that tells us much on its own. Additional problems are that a lower average time is better, which may be confusing, the raw time measure is linear which means that increases in performance become harder to see the larger they are, and the value is tied to the unit of time which we used.

One alternative is to divide an arbitrary scalar value by the mean time and report this as score. This produces an inverse so that large increases in performance are easier to see. It also abstracts from the time unit, and a suitable dividend can be used so that the result is large enough to be reported as an integer.

Another alternative is to always report mean iteration time relative to some other reference implementation, or maybe a reference configuration of the same implementation. This suits our purposes in this thesis because we can report the performance of our implementation and others against the standard implementation of Ruby, MRI, and we can report the performance of different Ruby functionality or implementation techniques relative to each other.

The speedup S_{SR} for a subject implementation I_S with mean average score time $\bar{T}_S$ against a reference implementation I_R with mean average score time $\bar{T}_R$, is:

$$S_{SR} = \frac{\bar{T}_S}{\bar{T}_R}$$

Note that a speedup of 1 indicates that the subject implementation ran at the same speed as the reference implementation.

Summarising Across Benchmarks

This single scalar value representing the performance of an implementation across multiple benchmarks is sometimes called the composite score, or composite speedup.

We use the geometric mean average to summarise speedup across multiple benchmarks [1]. The composite speedup C for n benchmarks with speedups S_1

to S_n is:

$$C = \left(\prod_{i=1}^{n} S_i \right)^{1/n}$$

Note that it is debatable how useful the composite speedup is as a metric. Individual benchmark scores or relative speedups are an understandable measure of how fast that benchmark runs, but a composite score or speedup is not understandable it applies to a theoretical composite benchmark.

For this reason we only report composite scores as a general guide in limited cases, and do not perform further visualisation of composite scores, such as the error discussed below. Therefore these sections will only refer to the earlier arithmetic mean.

Error

The mean average sample time is a scalar value, but we have already said that some variation in individual sample times is to be expected. The *error* is the how large this variation from the mean average to individual samples is. We want to report that error for three reasons. First, an implementation that achieves a high mean average peak performance but an extreme variation in sample times may be less useful than an implementation with a slightly lower but more consistent mean average. Some of your customers receiving responses to request in $1s$ might not be worth it if others take $100s$. Secondly, we are trying to use these statistics to say with a degree of confidence that one implementation or technique has a higher peak peak performance than another, so we need to distinguish between two means that may appear to be different but really the range of samples overlaps significantly, giving us low confidence that one mean is usefully above another. Finally, a high error may also be an indication that peak performance has not yet been reached.

There are multiple ways to calculate and report an error. We can report the sample standard deviation, which is the square root of the variance of our samples:

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (T_i - \bar{T})^2}$$

We can also report the standard error of the mean, which is:

$$SE_{\bar{T}} = \frac{s}{\sqrt{n}}$$

We did not employ more sophisticated statistical significance testing such as confidence intervals or a t-test as is common in the natural sciences, as these often rely on samples being independent. As we have already said, this is not the case in our experiments.

Visualisation

The main tool we use to visualise speedup against a reference for a range of implementations or configurations is a bar or column graph. We usually include a bar for the reference implementation, of height or width 1.

We can show error by drawing error bars that extend above and below the top of the bar to show the value of whatever error metric we are using. Informally, we can say that if the error bars of two implementations do not overlap then the result is likely to be statistically significant.

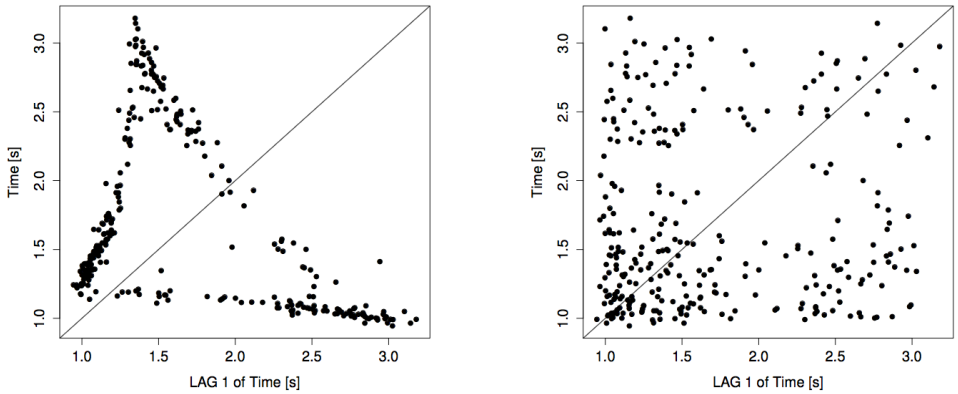
In some cases (see Chapter 6) differences in performance may be extreme — up to several orders of magnitude. In these cases we employ a logarithmic scale.

4.3.2 Advanced Statistical Techniques

Lag Plots

A lag plot shows for a set of samples T the points $(T_i, T_{(i-h)})$ where h is some period such as 1 or 4. The distribution of points across the plot shows you patterns in the sample for one iteration compared to some previous iteration. If there is no discernible visual pattern in the points then there is likely to be little correlation between one point and another, and we are more confident that the system is stable and individual samples are independent of each other. If a visual pattern can be identified then there may be an issue that should be examined such as insufficient warm up. However, there are also natural reasons why there may be a visual pattern – some benchmarks just appear to be cyclic in nature, not matter how long they are given to warm up.

We are not aware of any technique to automatically examine a lag plot and determine if there are patterns which may be of concern, so this is a manual task.



(a) Lag plots show how one measurement (x-coordinate) depends on its preceding measurement (y-coordinate). Any pattern in the plot suggests a strong dependency (left figure); a more random scatter indicates no dependency (right figure).

Figure 4.1: Example lag plots [59]

Figure 4.1 shows an example pair of lag plots from Kalibera et al [59]. The plot on the left shows a clear visual pattern so may need further investigation to check that there is sufficient warm-up. The plot on the right shows less cause for concern.

We did not construct and manually examine a lag plot for all experimental results as this would have been prohibitively time consuming. However, we did consult lag plots $h = 1$ while benchmarks and experiments were being developed.

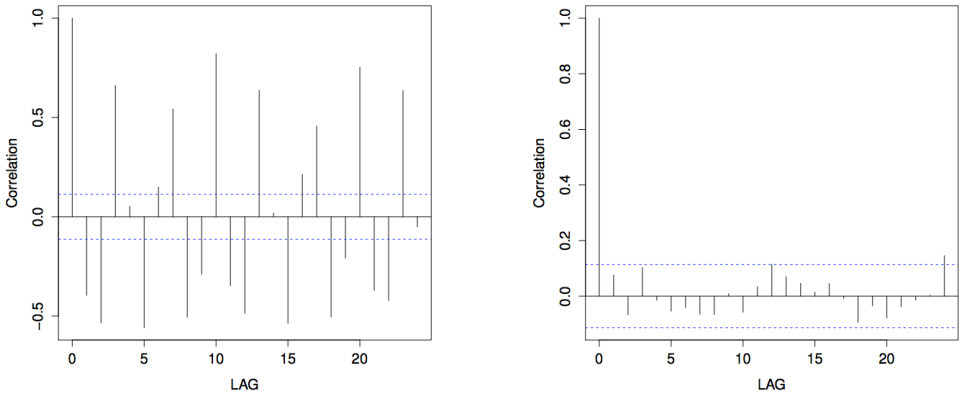
Autocorrelation Function Plots

A lag plot shows a single value of h which must be either picked arbitarily, or multiple plots generated to see a range of possible values for h. An ACF plot, also sometimes called *correlograms*, visualises a range of values for h by summarising the lag points with the correlation function, which is a measure of the strength of the relationship between a pair of samples.

Figure 4.2 shows an example pair of ACF plots from [59], again showing one that invites further investigation and one that does not.

The Kalibera-Jones Method

Kalibera and Jones present a reasoned and detailed approach to benchmarking in systems research [59]. Their technique is not a precise algorithm to be followed for benchmarking, but rather a set of recommended techniques, including some of the advanced techniques that we have discussed such as lag and ACF plots and



(b) ACF plots show correlation against the x^{th} preceding measurements. Large correlations (outside the dashed lines) or any pattern in the correlations marks a dependency (left figure). In contrast, the right-hand figure shows no dependency.

Figure 4.2: Example ACF plots [59]

multi-layered iterations. It still does not attempt to provide a single uniform way to benchmark, and its application still requires reason, oversight and judgement from the practitioner.

A key focus of the Kalibera-Jones method is a systematic process to determine how many iterations of the benchmark to run at each level, and how many of those iterations to discard. This is called a dimensioning experiment and will involve manual steps for each pair of benchmarks and implementations. To do this they take initial measurements and calculate an estimation of how much each level of iteration contributes to overall variability in results. Part of the motivation for this is to reduce wasted iterations caused by iteration counts that are sufficient but excessive. Dimensioning should be repeated from scratch for any change made to the implementation (or benchmark or the benchmark hardware), which certainly makes it unrealistic for benchmarking in continuous integration.

Kalibera and Jones also recommend an alternative metric to the errors already described: a *bootstrap confidence interval* that is closer to the confidence intervals and significance tests commonly used in the natural and physical sciences. These techniques often rely on samples being independent and identically distributed, as for example the weight of rabbits sampled from a given field are. As we have already said, this is not the case in the kind of research that we are doing. The bootstrap technique uses a simulated version of the experiment by random resampling of the sample population.

During the production of this thesis, the first research attempting to independently apply the Kalibera-Jones method was published. As part of this work

they implemented a Python package for the statistical routines involved in the method which is used in some experiments. The author of this thesis contributed a Ruby implementation of the same routines.

We applied some of the recommended techniques from Kalibera-Jones, but not the approach to multilayered iterations and experimental dimensioning. Part of the reason that we did not invest more resources into using these methods (as well as limited implementation time and limited compute resources) is that, as they point out, their method is designed to tackle the problem of distinguishing between small variations in performance. A median performance increase reported as a successful result in the systems field is just 10%. However in this thesis, the majority of our results are much more substantial than that. Therefore the more precise techniques are not as necessary.

4.4 Completeness

In Chapter 3 we used one other metric when talking about Ruby implementations: their *completeness*. Ruby is an extremely complicated language with a large builtin core library (builtin in the sense that it is normally provided by the implementation rather than as Ruby code that any implementation could run). This contains many routines that are rarely used but omitting functionality could allow implementation shortcuts to be taken that are not sound.

In early development of JRuby+Truffle we tackled this issue head-on by working through a list of difficult parts of Ruby that should be implemented before making performance claims, that was independently produced by a third party [77] and we have summarised here. Where reference is made to using functionality available in Truffle and Graal, the key publications for more details are Wüurthinger et al [115, 114].

`Fixnum` to `Bignum` promotion Ruby operators will automatically promote an operation on fixed-size native `Fixnum` integers that overflow to produce an arbitrarily sized `Bignum` value instead. This should be as simple as a `jo` machine instruction or architecture-specific equivalent which can jump to a slow path on overflow, but implementations using a conventional JVM or written in standards-compliant C++ may not have access to a primitive that does this. Truffle and Graal do provide such a primitive with the `ExactMath` class, which throws a

slow-path exception on overflow.

Floating point performance In dynamic languages, where values need to maintain their own type information, simple values such as integers and floating point numbers often need a heap-allocated box structure to store both the value and type pair. A common solution to this is tagging, where low bits are reserved in words which can identify them as either a pointer to a full structure which contains type information, or an immediate value which can have the low bits shifted away to be then be used without further indirection and no need for an allocated pair. Floating point values are often not tagged, as the logic is more complicated and the technique only became possible on 64 bit architectures, and so can be slower in implementations. JRuby+Truffle does not use tagging, as a technique to do this on a conventional JVM is not known. Instead we use a variant of storage strategies [17] and boxing combined with powerful partial escape analysis to remove unnecessary boxes.

Closures A closure is an object that represents the lexical environment (the local variables and sometimes other state) where a function was defined. Ruby's blocks are implemented using closures, and are used for much of the language's control structures. Closures can be difficult to implement, as they may require storing local variables on the heap instead of the stack, which involves heap allocation and indirection on access. JRuby+Truffle uses the existing functionality in Truffle and Graal to always logically allocate frames on the heap, but to use partial escape analysis to put them back onto the stack until they really need to be stored on the heap.

Bindings and eval Closures can be converted into a full Ruby object known as a `Binding`. This further complicates the implementation of closures, because now they are subject to the full range of dataflow available to any Ruby object such as storage in an another object or collection. They are also subject to metaprogramming methods such as `local_variable_get` and `local_variable_set`, so that names in a binding cannot be statically determined. Most importantly, a `Binding` object can be obtained from any block dynamically, so even if it statically looks like a closure does not escape and can be allocated on the stack, it may escape later on. The existing functionality for frames in Truffle and Graal

already allow for on-demand materialisation of stack frames onto the heap, via dynamic deoptimisation.

`callcc` and continuations JRuby+Truffle, like JRuby, Rubinius and Topaz, does not implement continuations and the call-with-current-continuation `callcc` method that would be familiar to Lisp programmers. This method is anyway deprecated in MRI. Continuations are fundamentally at odds with the JVM's specified execution model. A patch does exist to add non-standard support to HotSpot [100] but we have not explored using this yet.

Fibres Fibres are cooperatively scheduled threads. A fibre must decide to release control to another fibre, where a normal thread is usually pre-emptively forced to release control. A coroutine is a similar concept, where the release of control is through multiple returns from a subroutine. JRuby+Truffle, like JRuby, implements fibres using threads and blocking queues for cooperative scheduling.

Frame-local variables Ruby has a kind of variable which appears to be local to a frame, but can be implicitly set as a side effect of standard library methods. For example the variable `$_` appears to be frame-local, but it is set by the core library method `#gets` which reads a line. This means that implementations of Ruby need to have variables stored in one frame, but set by another method. JRuby+Truffle uses functionality from Truffle and Graal to access local variables in frames earlier in the call stack. As long as methods are inlined, and we deliberately inline methods which we know statically need to access local variables from earlier frames, then there is no cost for this.

C extensions Chapter 8 defines and discusses the implementation of C extensions.

String encodings Unlike other similar languages such as Python, Ruby has not standardised on an encoding of Unicode such as UTF-8 or UCS-6 as a universal encoding for internal representation[3]. Instead, strings contain raw bytes and also a reference to the encoding of those bytes. This causes the problem that languages

[3]It has been suggested that the Ruby developers did not standardise on Unicode due to technical disagreement with some of the decisions Unicode made on how to represent Asian characters, such as the inability to round-trip some other character sets through Unicode [60].

which do use an encoding of Unicode internally, such as Java, cannot correctly represent Ruby strings. JRuby+Truffle re-uses the JRuby solution to this, which is to represent strings simply as a `byte[]`.

Garbage collection JRuby+Truffle, and both Truffle and Graal, reuse the unmodified garbage collector from the JVM.

Concurrency and parallelism This thesis does not discuss concurrency or parallelism, with the exception that Chapter 7 talks about coordinating and communicating between different threads for runtime tasks such as attaching a debugger.

Tracing and debugging Chapter 6 defines and discusses the implementation of tracing and debugging.

`ObjectSpace` Chapter 7 defines and discusses the implementation of `ObjectSpcae`.

Method invalidation Method invalidation refers to the problem of efficiently detecting when a method has been monkey-patched or for some other reason a previously cached method for a given class may longer be valid. In JRuby+Truffle this is implemented using assumptions as described in depth in Chapter 7, and dispatch chains as described in Chapter 5.

Constant lookup and invalidation Constant lookup, caching and invalidation is done in the same way as method caching and invalidation in JRuby+Truffle.

Ruby on Rails JRuby+Truffle does not yet run Ruby on Rails. This is due to completeness of the core library and large libraries such as OpenSSL and database drivers, rather than any missing language functionality. At the time of writing, the first layer of the Rails library stack, called Active Support, is mostly working.

Later in development we were able to run test suites to verify that our implementation is correct. The completeness metric we use is the percentage of the RubySpec [95] set of unit tests (this particular type of tests are called specifications in the Ruby community but they are not any kind of formal specification).

4.5 Benchmarks Used

There is no single unified Ruby benchmark suite, comparable to industry benchmarks such as SPEC CPU and SPEC JVM or scientific benchmarks suites such as SciMark. Porting one of these benchmark suites to Ruby would not be representative of real Ruby code in any way. MRI and other implementations include a set of benchmarks of key operations, and some slightly larger micro benchmarks.

One reasonable position to take is that the only useful benchmark is the application which you are actually interested in running. This may be useful to industry practitioners if they are comparing multiple implementations of the same language and all implementations are capable of running their application. However it is not useful to industry practitioners who want to compare performance of implementations of multiple languages, as they would have to implement their entire system in each language, and it is not useful to early implementations of an established language such as JRuby+Truffle or Topaz, which are not yet able to run any complete applications. It is also not useful to language implementers, as they do not have a single application they are interested in.

The other extreme is synthetic benchmarks. These are very small applications, generally a screenful or two of code, that performs a single simple task. A key advantage of these benchmarks is that they are simple. For example, JRuby+Truffle was able to run the fannkuch benchmark within a few weeks. Another advantage is that they are typically very well understood by researchers which allows new implementations to be quickly tuned to run them well, but they are highly unrepresentative of real Ruby code. For example, nobody is making money by running a web service in Ruby to provide solutions to n-body problems.

In between the two extremes is the option to take some representative production applications, or libraries, and run parts of them as benchmarks. These can still be easy to understand if they are small enough, but they will likely stress very different parts of the language and library. In our case, we found that synthetic benchmarks often tested simple object field access, floating point arithmetic, and array accessing. The production benchmarks that we found instead tested functionality that is much more idiomatic to Ruby such as large numbers of small intermediate higher-level data structures such as hashes and more core library routines.

4.5.1 Synthetic Benchmarks

The first set of benchmarks we use are those from the Computer Language Benchmarks Game, formerly known as the Shootout benchmarks [31]. This includes classic benchmarks such as Mandelbrot, fannkuch [11] and n-body. We also used the key benchmark from the Topaz project, a neural network.

A complete list of synthetic benchmarks with a description and qualitative properties of each can be found in Appendix B.

4.5.2 Production Benchmarks

The second set of benchmarks we use are kernels from two Ruby libraries that we have already introduced: Chunky PNG [107] and PSD.rb [69], and their native equivalents Oily PNG [67] and PSD Native [68].

We identified methods in these libraries that are performance critical for real applications by looking at those that have been replaced with native versions in the native equivalents of the libraries. We took each of those methods and created a harness to exercise them, mocking out functionality unrelated to peak performance, such as IO.

Examples of kernels include resampling an image, extracting a single colour channel from a packed pixel value, replacing one image with another, encoding and decoding run-length-encoded data, converting between colour spaces and several different implementations of compose functions in the Photoshop file format.

A complete list of production benchmarks with a description and qualitative properties of each can be found in Appendix B.

4.6 Benchmark Harnesses

Our benchmark harness has two components. This split is necessary to have a simple component run by implementation under evaluation, as some implementations are not complete enough to do tasks such as reading and writing files.

The benchmark process itself runs a simple loop of benchmark iterations, recording the time at the start of the iteration, and reporting the elapsed time afterwards on `stdout`. Time is measured using a high performance monotonic timer if the implementation provides one. Topaz does not, so user time is used which is at risk of effects such as jumping (the clock jumping forward or backward

in time such as to correct for drift in comparison to a reference time source such as an atomic time service) and smearing (the clock speeding up or slowing time for a period as an alternative way to correct the same issue). The benchmark process then listens on stdin for a command to either stop or continue with another iteration. The benchmark process itself does not have any logic for warmup or any other techniques.

The benchmark process is run as a child process of a driver process. We do not believe that the extra process causes any unfair interference, as a system will typically be running many processes anyway, and communication between the processes is done outside the region of code which is timed. This process determines when warmup is complete using a simple technique. The driver runs the benchmark for at least I_{grace} iterations or at least T_{grace} seconds (whichever comes last) before considers warmup. The driver runs the benchmark in warmup mode until the range of the most recent I_{window} of the T samples, relative to the mean, is less than some value r, $\frac{range(T)}{\bar{T}} < r$. When those I_{window} samples are seen, they become the first I_{window} measured iterations. If the benchmark does not reach this state within I_{max} samples or T_{max} seconds (whichever comes first), a warning is printed and the benchmark is considered to have not reached a steady state but starts measuring anyway. We then take the first $I_{measured}$ samples, starting with the I_{window} samples we already have.

We set $I_{grace} = 30$, $I_{window} = 20$, $I_{max} = 100$, $I_{measured} = 100$, $T_{grace} = 30s$, $T_{max} = 240s$, $r = 0.1$. These numbers are arbitrary, but they are generous and in practice allow for sufficient warmup time for all implementations, without running too many iterations for the much slower implementations which reach steady state very quickly as they do not use any dynamic optimisation.

The benchmark driver tells the child process to stop when it has enough measurements. The driver stores all samples, performs the statistical methods, and then runs the next benchmark or implementation. When all the data requested is gathered, a human-readable interactive HTML and JavaScript report is generated, along with raw data output in a variety of formats for visualisation.

4.7 Repeatability of Results

Repeatability means a third party taking the same experiments as the original experimenter and running them to verify that the same results are achieved. An

example of repeatability would be a third party taking your implementation, benchmarks and configuration and running them on similar hardware to check that they see the same results as you do.

This is different from the higher standard of *reproducibility*, which is addressed later.

In the computer science systems research community, repetition of research is not common, and can produce controversial results when it is attempted.

The experiments in Chapter 5 were submitted to the publication venue's artefact evaluation committee in the form of source code, benchmarks, configuration, and a virtual machine image. Feedback from the committee showed that they were able to successfully repeat our results, and awarded the work an 'approved' badge.

The experiments in Chapter 6 and 7 were made available to the reviewers in a similar manner, but the venues in which the work was published [92, 23] did not have a formal artefact evaluation stage.

Experiments in other chapters either were not possible to disclose to reviewers as the implementation is proprietary, or were not submitted as the venue did not have an artefact evaluation stage.

4.7.1 Hardware

Based on the expected working environment and goals we described earlier in this chapter, our experiments were conducted on enterprise-grade server hardware with generous memory capacity.

The majority of experiments in this thesis were conducted on a system with two Intel Xeon E5345 processors based on the Core micro-architecture with four cores each at 2.33 GHz and 64 GB of RAM, running 64bit Ubuntu Linux 13.04, with kernel 3.8.8.

Experiments in Chapter 5 were conducted on a system with two Intel Xeon E5520 processors based on the Nehalem micro-architecture with four cores each at 2.26 GHz and 8 GB of RAM, running 64bit Ubuntu Linux 14.04, with kernel 3.11.

In both cases we used the distribution's default versions of software such as compilers and linkers.

4.7.2 Source Code

The primary research artefact resulting from this thesis is the JRuby+Truffle implementation of Ruby. Source code is licensed as part of JRuby under the EPL 1.0, GPL 2 and LGPL 2.1 licenses.

Revision `bbdd0ab`[4] of JRuby was used to obtain the summary results in Section 3.3.5. The latest version of JRuby is also available from GitHub[5].

Individual chapters used revisions of JRuby at the time the work was done. As JRuby+Truffle is an evolving system and some techniques have developed since the experiments described in this thesis, there is no single revision of JRuby that contains all the implementation work in this thesis, although all the ideas themselves are still a key part of above revision in some form.

Chapters 8 describes experiments that include proprietary implementation that we are not able to make available for third party examination at this stage.

4.7.3 Benchmarks

The benchmarks and the harness described above are available as the *bench9000* subproject of JRuby. Source code of the harness is licensed as part of JRuby under the EPL 1.0, GPL 2 and LGPL 2.1 licenses. Individual benchmarks are available under their own (generally permissive) licenses.

Revision `f7d26949bf`[6] of bench9000 was used to obtain the summary results in Section3.3.5. The latest version of bench9000 is also available from GitHub[7].

4.7.4 Reproducibility of Results

Reproducibility is a higher standard than simple repeatability, and refers to the same idea being applied and evaluated, from scratch, by a third party. An example of reproducibility would be a third party taking a published research paper and writing their own implementation and benchmarks and running on their own hardware to check that they see the same results as the original experimenter.

Verification of reproducibility in the computer science systems research community is even less common than repeatability.

[4] bbdd0ab6028aa51f1edc0d0768038a7f96b5de22
[5] https://github.com/jruby/jruby
[6] f7d26949bf6f1ce77659b3b3cd5660b563073898
[7] https://github.com/jruby/bench9000

Of the work in this thesis, we can say that the ideas in Chapter 5 have been effectively reproduced by a third party, as there was an independent simultaneous implementation of dispatch chains in both the JRuby+Truffle and TruffleSOM projects. This has shown that dispatch chains are applicable to both Ruby and Smalltalk, and that the performance they achieved can be demonstrated by two researchers with two independent implementations.

Work in other chapters has not been reproduced, but there has been a degree of verification now that the ideas are being implemented in other Truffle languages. For example, the debugging work described in Chapter 6 is now being applied to the Truffle JavaScript and R implementations.

4.8 Summary

In this chapter we have set the goal that will be used for many of the experiments in this thesis as minimising the real wall-clock time to solve a computation, and so set our metric as the the peak temporal performance after an initial period of warmup.

We have described how we assess the completeness of an implementation of Ruby and how we use this to show that JRuby+Truffle supports almost all of the Ruby language and core library and is not a trivial implementation missing features that may invalidate our work.

We introduced the benchmarks that we have used, with more information available in Appendix B, and described our benchmark harness.

Finally we have shown that parts of the work in this thesis have been subject to verification through repetition and reproduction.

The novel techniques, benchmarks and harnesses developed for this thesis have recently been applied by two independent groups of researchers for work on ahead-of-time compilation of Java [105] and a project to re-purpose components of a VM [102].

Chapter 5

Optimising Metaprogramming in Dynamic Languages

5.1 Introduction

Conventional approaches to optimising dynamic programming languages, such as those described in Section 3.1 including polymorphic inline caching, dynamic optimisation and deoptimisation, have largely solved the problem of implementing dynamic language features such as dynamic typing and dynamic dispatch. However metaprogramming has not received the same research attention and is often not optimised, even in mature language implementations. For example on the HotSpot JVM reflective method calls using the `java.lang.reflection` API has an overhead of around 6× [73].

In Chapter 2 we introduced metaprogramming features of the Ruby language and showed how they are used in patterns in performance-critical parts of the Ruby ecosystem. In Ruby, probably more so than in other languages, metaprogramming should not be viewed by implementers as a side-channel that does not need to be optimised, but instead as just another form of dispatch.

This is the key contribution of this chapter: a generalisation of existing polymorphic inline caching techniques that are polymorphic in the name of the method as well as the type of the receiver, that we call *dispatch chains*. We use dispatch chains to implement Ruby's metaprogramming features including dynamic sends, testing if a class responds to a method, and the `method_missing` hook for methods that a class does not respond to.

We demonstrate the impact that this has on a sub-set of the benchmarks

described in Chapter 4 that use metaprogramming extensively.

The research contributions in this chapter were an independent piece of work, initially presented informally in a blog post on the author's website in July 2014[1]. Stefan Marr, a researcher at Inria, independently developed a similar technique for use in his implementation of the Smalltalk language. Together we produced a single joint publication [73], which covered the technique we had both independently developed and evaluated it in the context of both of our language implementations.

5.2 Metaprogramming

Metaprogramming is an informal term which means any kind of programming that treats the program as normal application data in some way. In this thesis we are primarily interested in the kind of metaprogramming where method calls are made with a method name as application data, or other similar operations. Other forms of metaprogramming available in Ruby include operations such as redefining 'constant' values, and in other languages metaprogramming is much more advanced and may allow the program itself to be modified in some way, such as with a macro system.

In Chapter 2 we talked about a common workaround for the low performance of metaprogramming in most implementations of Ruby, which was to programmatically generate new methods. We also don't consider this form of metaprogramming in this chapter, as it is our goal to optimise the dynamic method calls themselves, rather than work around them.

5.3 Existing Implementations

Existing implementations of Ruby generally implement metaprogramming operations with a separate mechanism as is used for normal calls. Implementations such as Rubinius and JRuby have complex systems for optimisation of inline caching but they are not used for metaprogramming.

Listing 5.1 shows the JRuby implementation of the Ruby `send` method. The call to `searchMethod` on line 4 is only subject to per-module caching, which is implemented as an expensive lookup in a `ConcurrentHashMap` object. There is

[1] http://chrisseaton.com/rubytruffle/pushing-pixels/

```
public IRubyObject send(ThreadContext context,
      IRubyObject arg, Block block) {
    String name = RubySymbol.objectToSymbolString(arg);
    DynamicMethod method = searchMethod(name);
    return method.call(context, self, this, name, block);
}
```

Listing 5.1: JRuby's implementation of `send` (simplified).

```
Object* Object::send(STATE, CallFrame* caller, Symbol* name,
    Array* ary, Object* block) {
  LookupData lookup(this);
  Dispatch dis(name);

  Arguments args(name, ary);
  args.set_block(block);
  args.set_recv(this);

  return dis.send(state, caller, lookup, args);
}
```

Listing 5.2: Rubinius's implementation of `send`.

no caching of the method for this particular call site, and so no inlining. Part of the reason for this is that the `send` operation is implemented as a Java method, which provides no facility for storing state between calls.

Listing 5.2 shows the similar Rubinius implementation, which also does no caching for the call site as it has no way to reference the call site. The Rubinius code has an additional problem in that this is a compiled C++ method, so the LLVM optimiser that is applied to the Ruby bytecode cannot reason about the behaviour of this method and cannot inline it.

5.4 Dispatch Chains

Our solution to the problem of optimising metaprogramming sends in Ruby is to generalise the idea of a PIC into something that we call a *dispatch chain*. A PIC is usually a linear set of cache entries with the class of the receiver as the cache key, and the method to call for that receiver, for a given name, is the cache value.

Figure 5.1 shows the shape of this conventional PIC. The receiver class is

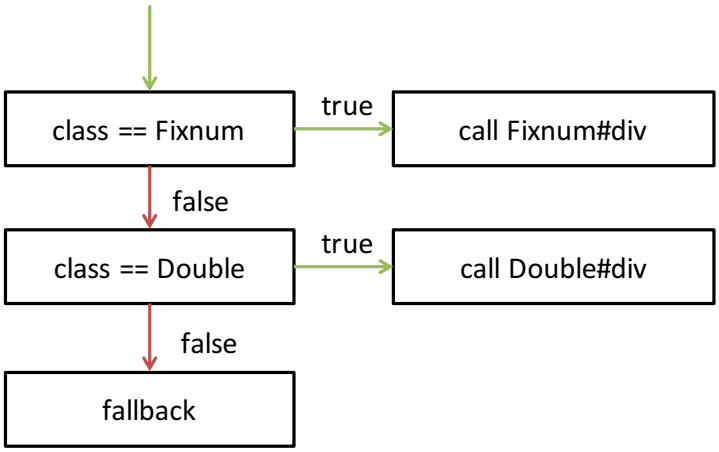


Figure 5.1: A conventional PIC.

compared against the cached classes and when they match the resulting method is called, or possibly inlined. Fallback cases will update the cache with a new entry, using dynamic deoptimisation if machine code has been generated.

A logical extension of the idea of PICs therefore could be to have nested caches. An initial cache could map from method names to a set of second-level caches, which map from receiver classes to methods. This is the basic idea of a dispatch chain.

There are multiple configurations that could be used for a dispatch chain. In the examples given so far, we varied on the method name first, and the class of the receiver second. The advantage of this approach is that the method name is usually constant so checking for it first means that there only needs to be one comparison in most cases. In Ruby, the `String` object is mutable, so comparing strings for equality is not always a cheap operation. Another configuration could be to vary the receiver class first, and the method name second. If the name is usually constant then this approach may cause there to be multiple copies of the name guard, with one for each method.

Figure 5.2 shows an example of the former, name-first dispatch chain, and Figure 5.3 and example of the latter, receiver-class dispatch chain. If the case where the name does not vary, it can be seen that in the first configuration only one check against the name is done, but in the second configuration there are two–one after each receiver-class cache passes.

With a strong optimising compiler such as Graal, we may not need to worry

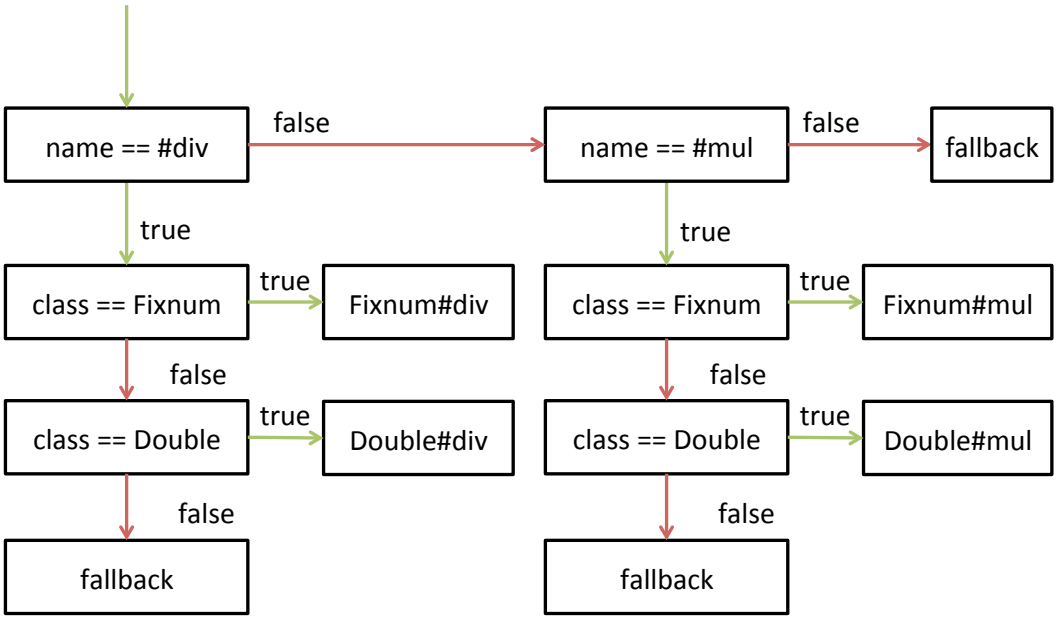


Figure 5.2: A name-first dispatch chain.

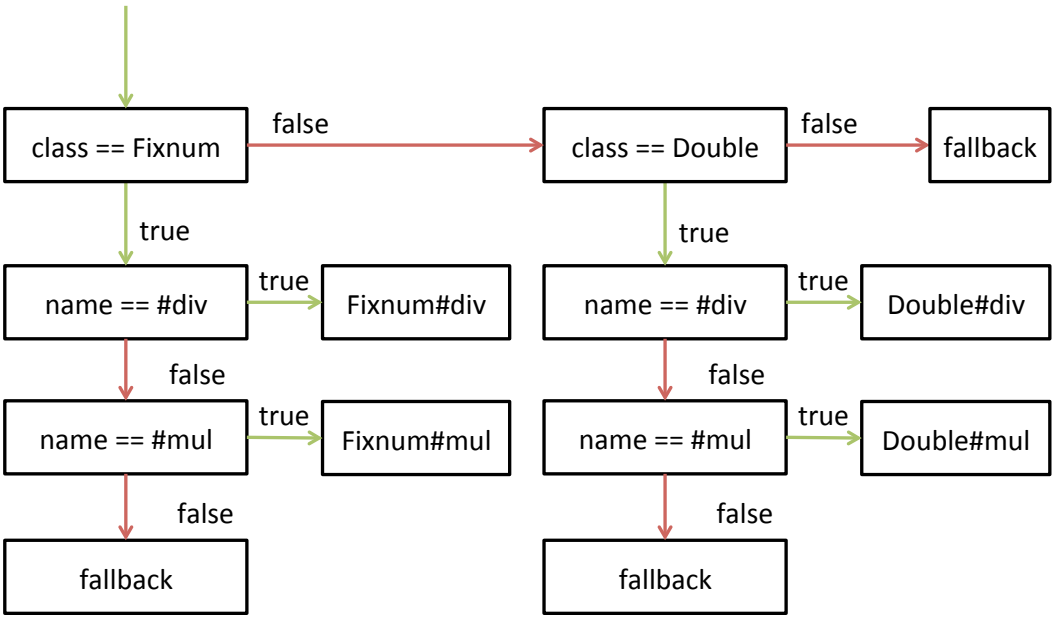


Figure 5.3: A receiver-class-first dispatch chain.

about the difference between these two configurations, as any repeated guards may be able to be de-duplicated and hoisted to where they are common between multiple branches.

A third configuration option linearises the dispatch chain and has each cache entry as a tuple of the method name and the receiver class. The advantage of this approach is that it is simpler with no nested chains. The disadvantage is that if there is a single method name it will get checked again for each receiver class. Again, we may be able to rely on the optimising compiler to de-duplicate these checks.

In the following evaluation, we used the first configuration and varied on the method name first and the class of the receiver second. After the evaluation was complete and the research published we have subsequently switched to the third configuration where we vary on a tuple of the method name and the receiver class at the same time. This was done for simplicity of implementation (as the data structure is linear again) but a detailed evaluation for the relative performance of these different configurations still needs to be done on large real-world applications, and ideally could be made available as a tuning option.

Another open question is whether it is worth sorting the chains in any way, such as based on how frequently each one is used, or if a simpler approach such as always append or always prepend new entries would be better. In our approach we always prepended. This handles the case where an initial call that is never used again does not always have to be guarded against.

5.5 Implementation

Our second contribution in this chapter is to observe that the structures we have described above are trees, and so can be implemented using the same nodes as we use to implement the Ruby AST. Each condition in the chain is implemented as a node with a child node for the cache-miss case, and a child node for the cache-hit case, in the case of first level of caching, or a reference to the method to be called in the case of the second level of caching.

Listing 5.3 shows simplified nodes representative of our implementation. The name chain node caches a name and a child chain to be used if that name matches, or another name chain node to try if it doesn't. The class chain node caches a class and a method to call if that class matches, or another class chain node to

try if it doesn't.

Not shown are nodes for the fallback cases. Again these are very elegantly expressed as AST nodes, as to modify the cache they can simply specialise themselves to be a new cache entry.

A key property of the implementation of dispatch chains in JRuby+Truffle is that it is the only type of method cache. Rubinius and JRuby both feature multiple caching mechanisms and attempt to use the simplest possible, but in JRuby+Truffle a full dispatch chain, that can handle a varying name, is always used, even when the name is constant. This is for uniformity and simplicity.

5.6 Application In Ruby

In JRuby+Truffle, any location that makes a call will use the single unified dispatch chain technique. In effect, all sends in JRuby+Truffle are metaprogramming sends of the form `receiver.send(:name, arguments)`, and we rely entirely on the partial evaluation in Truffle's Graal backend to remove the degree of freedom in the method name if we are not using it.

Listing 5.4 shows a simplified version of the AST node for a conventional call in JRuby+Truffle. Line 3 declares the name of the method that is being called as a final field. The name is set statically in the source code for the call site and never changes. Line 4 declares a dispatch chain node as a child (the `@Child` annotation). Line 6 is the standard AST execute method, which somehow executes child nodes to get the receiver and arguments (not shown). Line 9 then executes the dispatch chain node call, passing the method name, which was final as a runtime parameter. Therefore the method name is not a final value in the dispatch chain itself, but we have confidence in the partial evaluation phase of Truffle's Graal backend to propagate that constant and entirely constant fold and remove the logic in the dispatch chain that handles varying method names.

Listing 5.5 shows the JRuby+Truffle implementation of the `send` operation, so our version of the JRuby code in Listing 5.1 and the Rubinius code in Listing 5.2. Our core library methods are implemented as nodes, rather than Java or C++ methods, which means that they are an object and have a location for us to store an inline cache. We use a child `DispatchChainNode` in exactly the same way as we did in the node for conventional calls. The only difference between the use of the dispatch chain node here is that the method name is now an argument,

where in the call node it was a final field.

However, if the expression going into that argument happens to be a constant anyway, such as if it was a literal symbol, as is the case in an expression such as `receiver.send(:name, arguments)`, then the code is equivalent, and the generated machine code will be the same with the same performance. If the name is not constant then we will also compile the parts of the dispatch chain which deal with the varying name.

5.7 Evaluation

We measured the impact of using dispatch chains to optimise reflective operations in JRuby+Truffle using the eighteen image composition kernels from the PSD.rb library as benchmarks. These compose operations, which produce a single colour value from multiple inputs, are key for performance in image applications that use them as they are run for every pixel in an image. In PSD.rb these are implemented using multiple metaprogramming operations. Following the Ruby philosophy of choosing convenient implementations over creating extra abstraction with classes, the library developers chose to pass the name of the composition operation as an argument, which is then used by the reflective method invocation `send`. Within each composition operation, colour value manipulation methods are called that are not part of the object. These are caught via `method_missing`, filtered with `respond_to?` and delegated with `send`, in a form of ad hoc modularity. In an extreme case for each pixel in the image there are seven calls to `send` and six calls to each of `method_missing` and `respond_to?`.

This sequence of operations was summarised in the Acid Test benchmark described and evaluated in Subsection 2.3.6.

When these benchmarks are run each in a separate invocation of the VM, the dynamic method calls are monomorphic at the call site. That is, each call to `send` in the benchmark source code will only actually see one method name for the whole program run. This is a good example of metaprogramming being used to make the program simpler (from the perspective of the Ruby community) but the dynamism not actually being needed in practice. An important point to make is that we rely on Truffle's sophisticated *splitting* technique (Subsection 3.2.4) to create a new copy of the `send` method each time it is used in the source code. If this was not done then there would be a single call site for dynamic method calls

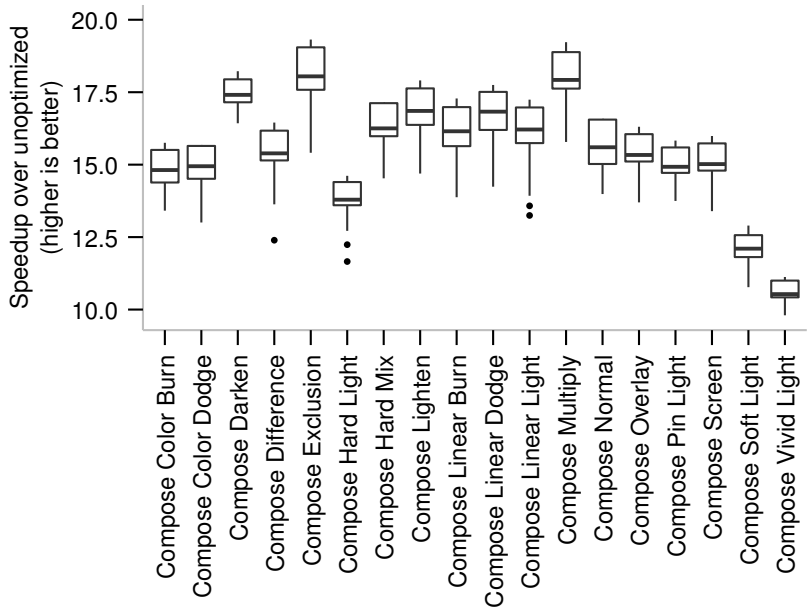


Figure 5.4: Metaprogramming speedup on compose benchmarks.

– that of `send` itself, which would very quickly become megamorphic.

Figure 5.4 shows the speedup produced by our dispatch chain technique on the compose benchmarks, relative to a non-caching technique, similar to that used by other implementations of Ruby. Dispatch chains give between 10× and 20× speedup over the conventionally optimised calls.

This form of graph shows a box plot, with the median, 25th and 75th percentiles, and whiskers at the maximum or minimum sample within 1.5 interquartile range. Outliers beyond this range are shown as dots.

5.7.1 Application in Smalltalk

Marr et al [73] independently and concurrently developed the same technique as we have described here, and applied it in the context of a reduced dialect of Smalltalk called the Simple Object Machine (SOM).

In these experiments, an implementation of dispatch chains was tried in the context of two high-performance programming language implementation frameworks. The SOM-MT results implement SOM and dispatching chains using the RPython meta-tracing JIT, as used in PyPy and the Topaz implementation of Ruby, and the SOM-PE results use Truffle and Graal.

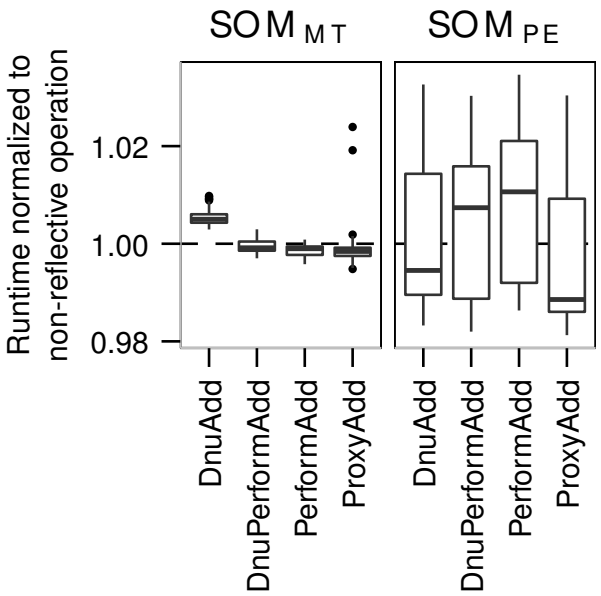


Figure 5.5: Overhead for metaprogramming in SOM using dispatch chains.

Figure 5.5 shows the results of an experiment at a smaller scale than the one we have just presented, using microbenchmarks that just test the overhead of the particular metaprogramming operation, compared to a conventional static operation. Each benchmark uses a counter that implements an increment method. The baseline for comparison calls the increment using a conventional static send. *PerformAdd* calls the increment method reflectively. In *DnuAdd* the counter does not implement increment and instead uses SOM's missing-method handler to do the integer increment. *DnuPerformAdd* combines missing-method handling with a reflective method call. The *ProxyAdd* benchmark combines missing-method handling with reflective calls to assess the overhead of dynamic proxies built with it. The data is noisy, more so for the implementation of SOM based on the JVM, but it shows that meta-programming operations can be implemented without overhead.

5.8 Summary

In this chapter we have shown why existing implementations of Ruby do not perform well in metaprogramming calls. We identified two problems in their approach.

First there was no caching technique that could vary on both a name and a receiver class. We solved this with dispatch chains, a generalisation on PICs, and we showed that the implementation of dispatch chains is particularly elegant in the context of a self-optimising AST interpreter.

The second problem in existing implementations was that there was no suitable location to store a cache for a particular call site, because both JRuby and Rubinius implement Ruby core library methods as Java or C++ methods. We solved this problem by making our core library methods nodes, which are objects in which we can store state. With inlining of ASTs by splitting (described in Chapter 3), there is an instance of this object.

We finally identified that in an implementation with dynamic optimisation it is possible to generalise to a single caching primitive, which is the dispatch chain, and rely on optimisations to remove the unused logic for a varying name if it is not used due to a constant name.

Subsequent results in Chapter 8 will look at the performance of the native code which is usually used to replace these metaprogramming operations.

The techniques in this chapter have given us a way to implement metaprogramming operations as efficiently as conventional operations. In fact, in JRuby+Truffle we replaced conventional operations with the same logic as metaprogramming operations. During peak performance, when the program is stable and has been optimised as described in Chapter 4, the technique has zero-overhead compared to conventional operations. There will however be overheads in compilation time and memory consumed by the compiler, but we were not attempting to reduce or maintain these other metrics at previous levels.

```
public static class DispatchNameChainNode {

    private final String cachedName;
    @Child private DispatchClassChainNode cacheHit;
    @Child private DispatchNameChainNode cacheMiss;

    public Object execute(VirtualFrame frame,
      Object receiver, String methodName, Object... args) {
        if (methodName.equals(cachedName)) {
          return cacheHit.execute(frame, receiver, args);
        } else {
          return cacheMiss.execute(frame, receiver, args);
        }
    }

}

public static class DispatchClassChainNode {

    private final Class cachedClass;
    private final Method cacheHit;
    @Child private DispatchClassChainNode cacheMiss;

    public Object execute(VirtualFrame frame,
      Object receiver, String methodName, Object... args) {
        if (receiver.getClass() == cachedClass) {
          return cacheHit.call(frame, receiver, args);
        } else {
          return cacheMiss.execute(frame, receiver, args);
        }
    }

}
```

Listing 5.3: JRuby+Truffle's implementation of dispatch chain nodes (simplified).

```
public class RubyCallNode extends RubyNode {

    private final String methodName;
    @Child private DispatchChainNode chainNode;

    public Object execute(VirtualFrame frame) {
        final Object receiver ...
        final Object[] args = ...
        return chainNode.call(frame, receiver,
          methodName, args);
    }

}
```

Listing 5.4: JRuby+Truffle's implementation of a conventional method call (simplified).

```
public static class SendNode {

    @Child private DispatchChainNode chainNode;

    public Object execute(VirtualFrame frame,
      Object receiver, Object methodName, Object... args) {
        return chainNode.call(frame, receiver,
          methodName, args);
    }

}
```

Listing 5.5: JRuby+Truffle's implementation of `send` (simplified).

Chapter 6

Optimising Debugging of Dynamic Languages

6.1 Introduction

The optimisation of a programming language implementation, and the ability to debug programs running on that implementation are often conflicting goals. Optimisations generally focus on removing redundant work and abstractions that are not needed so that the program is simpler and runs in less time. Debuggers may have needed this information in order to monitor and modify the running program. The most common solution to this problem is to disable optimisations when the user wants to use a debugger.

Static compilers for languages such as C often have optimisation levels that are set by the programmer, and optimisations to include debug meta-information into the produced binary. When the programmer anticipates needing to attach a debugger, they will normally disable the optimisations and enable the debug meta-information. If the programmer did not know ahead of time that they would want to attach a debugger, and their program is already running, the experience using the debugger is much worse, and it may not be reasonably practical to debug the program at all.

Even many dynamic programming language implementations have similar problems. As we will show in the next section, only the reference implementation of Ruby, MRI, has the full set of debug and introspection features always enabled. Alternative implementations such as JRuby and Rubinius disable debugging features unless a special debug mode is set at startup, or do not implement them at

all. As with a C program, programmers are expected to know ahead of time that they will want to debug a program in order to use these features.

Based on our assessment of how Ruby is used in industry in Chapter 4, we would like to let Ruby programmers debug long-running processes. The Ruby program should always be running in a state where a debugger can be attached, and should run with peak temporal performance until that happens. When the debugger is removed, the process should then return to peak temporal performance.

While the debugger is attached, we want to maintain peak temporal performance wherever possible. For example, if a line breakpoint is installed in a method, the method should still run at peak performance, until that breakpoint is triggered. If there is a line breakpoint with a condition that should be evaluated to see if the breakpoint should be triggered, then the overhead should be proportionate to the cost of that evaluating the condition. We do not want the program to enter a special state where optimisations are disabled whenever the program is being debugged.

As will be covered in more depth in the next section, this is an extremely high bar to set. No other implementation of Ruby achieves this, including MRI, which does not have optimisations to disable. In fact, even long-established programming language implementations that have been highly tuned for tooling such as the Java HotSpot JVM do not achieve the same performance for a method that has a breakpoint attached as one that does not [4].

The key contribution of this chapter is the novel concept of AST wrapper nodes for debugging. We begin by describing the current state of Ruby debuggers and language features designed to aid debugging. For simplicity we outline a small prototype debugger that we will implement to demonstrate our techniques. We then introduce the high-level concept of AST wrapper nodes in the context of a self-optimising AST interpreter. We show how wrapper nodes can be used to implement our debugger and other Ruby language features. We evaluate our technique against other Ruby implementations and show how for most cases it has *zero-overhead* for the case where a debugger is able to be attached, is attached, or has been attached, compared to a configuration where the debugger is disabled. In cases where work cannot be removed entirely we will show how it has reasonable overhead.

In this chapter we make the reasonable assumption that the optimisations

that our compiler make are correct, and so do not consider the problem of optimisations changing the meaning of a program.

The research contributions in this chapter were initially independent work that supported the Ruby `set_trace_func` language feature (described later). The applicability of the technique for implementing a debugger was discussed with Michael Van de Vanter at Oracle Labs and the technique generalised in collaboration with him to fit that purpose. The work was evaluated and written up in collaboration with Michael Van de Vanter and Michael Haupt, also of Oracle Labs for a joint publication [92].

6.2 Ruby Debuggers

We can consider two kinds of debugging features in the Ruby language. There are debugging features built into the language that can be accessed by the running program, which are a kind of metaprogramming, and there are more fully-featured debugging tools, which are often implemented using the language features and are similar to conventional tools such as GDB.

6.2.1 Language Debugging Features

In Chapter 2 we introduced some of Ruby's powerful metaprogramming language features such as dynamic sends, missing method interception and so on. An even more advanced feature is Ruby's `set_trace_func` method. This allows a Ruby `Proc` (effectively an anonymous function) to be installed as a callback to be run on various interpreter events. For example, it is called every time the interpreter reaches a new line in the program's source code.

Listing 6.1 shows a simple `Proc` being installed that prints the line number as the program runs. The `Proc` receives various information in parameters each time it is called by the implementation, including the type of event, the file and line where the interpreter currently is, an identifier of the current method, a binding which is an object representing the current lexical environment (a set of local variables) and the current class's name.

The default implementation of Ruby, MRI supports `set_trace_func` and there is no option to enable or disable it. MRI implements `set_trace_func` with a dedicated `trace` instruction in the bytecode that is always present. It checks a flag to see if tracing is enabled and if so calls the trace `Proc`. This means there

```
set_trace_func proc { |event, file, line,
    id, binding, classname|
  puts "We're at line number #{line}!"
}
```

Listing 6.1: An example use of Ruby's `set_trace_func` method

is always an overhead on performance because the instruction is always present, even if it does not do a lot of work without an installed `Proc`. There is another overhead when a `Proc` is installed, in making the call.

JRuby allows `set_trace_func` to be used only when a special `--debug` flag is set. This flag completely disables all optimisations and adds frequent checks to a flag, similar to the approach used in MRI. Work is underway using the new IR system in JRuby to implement `set_trace_func` using an approach similar to ours[1].

Topaz is the only other implementation of Ruby beside JRuby+Truffle that always has `set_trace_func` enabled and also achieves relatively high performance. Topaz declares the variable that holds the current trace `Proc` to be a *green variable* [16]. A green variable is one that should be the same every time a trace (see Subsection 3.2.5) is entered. When no trace `Proc` is installed, the variable is `nil` upon entering the trace and can be assumed to be `nil` throughout, meaning that the check at each line if there is a trace method installed is a constant. If a trace method is installed, the compiled trace will be found to be invalid when the green variables are checked, and will be recompiled. A downside of this approach compared to ours is that it is a *global* field in Topaz's top level object and so any new language requiring similar functionality would require another global field. Our approach is *localised*, with `Assumption` objects that can be declared anywhere in the program, not just at the top level.

Rubinius and MagLev do not have any support for `set_trace_func`. The optimisation techniques they have applied are not easily compatible with `set_trace_func`, which is an example of the conflict between performance and optimisation that we described at the beginning of this chapter.

`set_trace_func` is also used as a fundamental construct to implement debugging tools, as described below. It was historically used to implement tools such as

[1] See Chapter 7 and the discussion on `Switchpoint` for the underlying primitive they will use

coverage, until specific APIs were introduced to Ruby to improve performance.

6.2.2 Debugging Tools

The Ruby community is strongly in favour of different forms of testing, including *unit-testing*, *specifications*, *testing-driven-development* and *behaviour-driven-development*. Coupled with the Ruby REPL, IRB, where code can be tried and refined interactively, and the poor performance of what debuggers there are available (shown later), many Ruby programmers may not be routinely using a conventional debugger tool.

The Ruby standard library includes a simple debugger that is implemented using `set_trace_func`. The name of the library is *debug*, but for clarity we will refer to it as *stdlib-debug*. Listing 6.2 shows an illustrative example of the `Proc` installed by stdlib-debug and how it looks for the `line` event and calls a method to see if it should break on this line, and if so it calls a method to do that. Our prototype debugger, described below, is loosely based on this interface with some simplifications.

```
1 set_trace_func proc { |event, file, line, id, binding, klass, *rest|
2   case event
3   when 'line'
4     if check_break_points(file, nil, line, binding, id)
5       debug_command(file, line, id, binding)
6     end
7   end
8 }
```

Listing 6.2: stdlib-debug using `set_trace_func` to create a debugger (simplified)

As will be shown, performance of stdlib-debug is extremely poor. Providing a superset of the functionality of stdlib-debug, *ruby-debug* (also referred to as *rdebug*) is implemented as a C extension to MRI to reduce overheads. It uses internal interfaces in MRI which crucially do not require the per-line logic to be written in Ruby, which MRI executes very slowly. This library is the foundation for most debugging tools commonly used with Ruby, eg, in RubyMine.

As JRuby has limited support for C extensions, ruby-debug has been ported to Java as *jruby-debug*. In the same way as ruby-debug it uses internal interfaces to try to improve performance, but as with `set_trace_func`, using jruby-debug

requires the `--debug` flag to be set, which disables all optimisations.

Rubinius includes a debugger that is integrated into the VM, providing a debug protocol and user interface. Unfortunately, we found that the Rubinius debugger silently failed to work in combination with JIT compilation[2] so we were forced to disable compilation manually.

Topaz does not support any debugger. MagLev has a debugger, but it is highly integrated into the underlying Smalltalk VM so the interface presented is not what a Ruby programmer would understand, and, crucially, it will not compile a method with a breakpoint so performance is always going to be very limited.

6.3 A Prototype Debugger

We designed a simple prototype debugger with limited features to experiment with implementation on JRuby+Truffle. The debug operations it provides includes the ability to:

- Set a line breakpoint that halts execution.
- Set a line breakpoint with an associated *action*: a fragment of Ruby code that might be guarded by a *conditional expression* and which might halt execution or anything else.
- Set a data breakpoint on a local variable in some method that halts execution immediately after an assignment, which is implemented with a line breakpoint on each line where the local variable is modified with an action to examine the value.
- Set a data breakpoint with an associated *action*, as with a line breakpoint.
- Continue execution.
- Basic introspection of the program structure and current state such as examining the value of variables and reporting the halted position.

[2] We reported this issue along with test cases to demonstrate the problem (https://github.com/rubinius/rubinius/issues/2942) but have not received a response at the time of writing.

6.4 Debug Nodes

6.4.1 Wrappers

The central construct in the Ruby Truffle debugger is the *wrapper node* or simply *wrapper*. This is a Truffle AST node with one child that:

- is transparent to execution semantics,
- by default just propagates the flow of program execution from parent to child and back, and
- performs debugging actions when needed.

Note that the logic of this node can be implemented without overhead, which will be explained later on. Starting with an AST produced by a conventional parser, we insert a wrapper as the parent of the first node corresponding to each location where we may want to install some debug action.

Listing 6.3 shows Ruby code that increments a local variable `x` and decrements a local variable `y` while `x < y`. This code has three locations where we might want to set a breakpoint, and two locations where we might want to break on assignment of a local variable.

```
1 while x < y
2   x += 1
3   y -= 1
4 end
```

Listing 6.3: Example Ruby code

Figure 6.1 shows the AST of this code as produced by the parser. Figure 6.2 shows the same AST with wrappers inserted wherever the Ruby parser tells us that the line number has changed, to implement line breakpoints. Each wraps a single child node.

6.4.2 Debug Operations

Potential debugging operations are implemented in the form of wrappers at every location where the user might want to request a debug action. The wrappers are always added, whether or not a debug action is initially installed, and whether

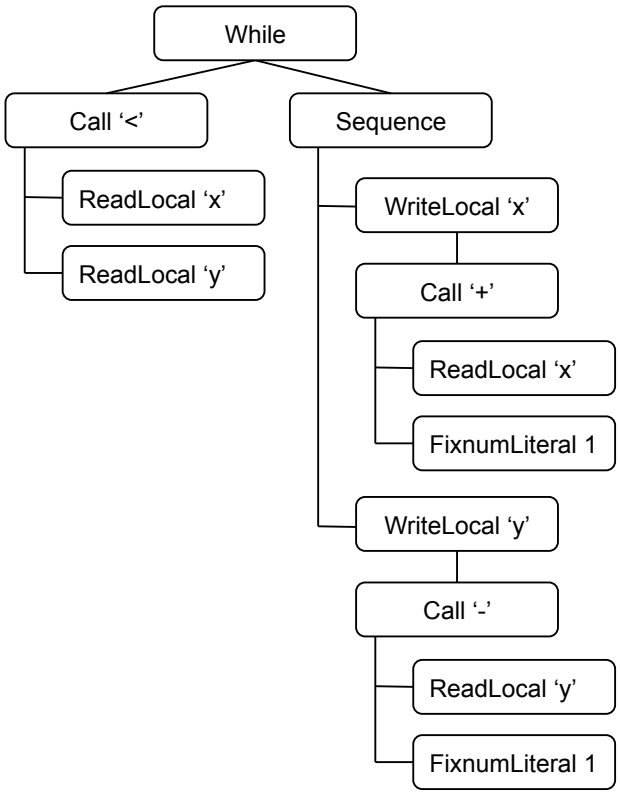


Figure 6.1: AST of Listing 6.3 without wrappers

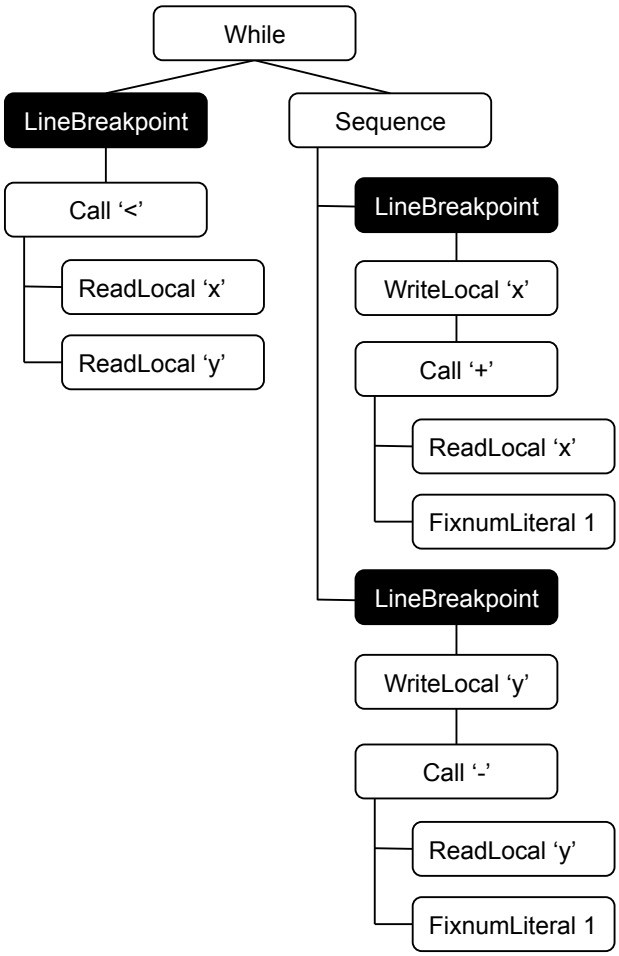


Figure 6.2: AST of Listing 6.3 with wrappers to implement line breakpoints

or not a debugging tool is currently attached. When added, the wrappers are initially in an *inactive* state. An inactive wrapper is simple: during each execution it only checks to see if it should be enabled, and if not propagates the flow of program execution to the wrapped node. The debug operation at a particular location can be enabled by replacing just the relevant inactive wrappers with *active* versions. The action that the active wrapper performs depends on the functionality it implements, and we describe several active wrapper nodes in section 6.4.4. When no longer needed, an active wrapper replaces itself again with an inactive version.

6.4.3 Assumptions

Many wrappers follow this pattern: a debugging node replaces itself with an alternate version when some presumably rare condition occurs. Truffle aggressively optimises code when given *hints* about what conditions should be treated as the normal case; instances of the `Assumption` class are one way to do this.

An `Assumption` is implemented as a `boolean` that is initially `true` until *invalidated*, at which time it becomes permanently `false`. For example, debugging code might create an instance to represent the fact that there is no breakpoint at a particular line of source code, and will only invalidate that assumption should a breakpoint be created.

Truffle applies important optimisations speculating that `Assumption.isValid()` always returns `true`. When an instance is invalidated (i. e., its value is set to `false`), Truffle `deoptimises` any method code that depends on that assumption (i. e., any code that calls `Assumption.isValid()` on the instance). Typically the program then replaces the node associated with the invalid `Assumption` and creates a new (valid) instance of `Assumption`.

6.4.4 Wrapper Roles

Our implementation of a Ruby debugger uses wrapper nodes to implement debug and metaprogramming functionality that is similar to that provided by other implementations.

Tracing

Ruby's core library method `Kernel#set_trace_func` registers a method to be called each time the interpreter encounters certain events, such as moving to a new line of code, entering or leaving a method, or raising an exception (Listing 6.1 shows an example). This method is used to implement other Ruby debuggers (such as the *debugger* library, detailed in section 8.10), profilers, coverage tools and so on. The trace method receives a `Binding` object that represents the current environment (local variables in lexical scope) as well as other basic information about where the trace was triggered. One trace method at a time can be installed, and it may be removed by calling `set_trace_func` with `nil`.

The Ruby Truffle debugger implements `set_trace_func` as an (initially inactive) *trace wrapper* at the location of each line. Each time it is executed, the inactive node checks the assumption that there is *no trace method* installed before propagating the flow of program execution. When the check fails, the node replaces itself with an active trace wrapper.

The active wrapper correspondingly checks the assumption that *there is a trace method* before first invoking the method and then propagating the flow of program execution. When the trace method has been removed, the check fails and an inactive wrapper is swapped back in. Using an `Assumption` object ensures that in the most common case the only overhead is the (inactive) wrappers performing the check.

Line Breakpoints

The line breakpoint and `set_trace_func` implementations are similar. However, instead of a single trace method, line breakpoint wrappers check if a method has been installed for their associated line of source code. The debugger maintains a map that relates source locations to `Assumption` objects. A newly constructed line breakpoint wrapper is given access to the `Assumption` that the current method for that line has not changed.

A triggered breakpoint halts program execution and starts an interactive session similar to the standard interactive Ruby shell, IRB. This debugging session runs in the execution environment of the parent scope at the breakpoint, so that local variables are visible in the debugger. Additional Ruby methods available in the shell include `Debug.where` (displays the source location where the program is halted) and `Debug.continue` (throws an exception that exits the shell and allows

program execution to continue). We have not yet implemented debug operations such as `next`, but believe these can be implemented with combinations of these techniques.

The action taken by an active line breakpoint node could be anything that can be expressed in Java (Truffle's host language) or, as with `set_trace_func`, a method written in Ruby. Listing 6.4 shows an example command to install a line breakpoint. This could have been written as part of the program, or typed into an interactive shell. The example prints a message to the log, but it could contain arbitrary Ruby code, including entry into the debugger.

```
1 Debug.break('test.rb', 14) do
2   puts "The program has reached line 14"
3 end
```

Listing 6.4: Example command to install a line breakpoint

Conditional Line Breakpoints

Conditional line breakpoints are a simple extension to line breakpoints. Since the breakpoint wrapper is a legitimate Truffle AST node, an `if` statement can be wrapped around the action that invokes the debugger. To support conditions written in Ruby, we can call a user-defined method to test the condition, in exactly the same way as we call a user-defined method in `set_trace_func`. Again, it is also possible to inline this method, so the condition becomes part of the compiled and optimised method.

Figure 6.3 shows the AST of Listing 6.3 with a line breakpoint installed on line 3 that contains the condition `y == 6`. The condition forms a first-class part of the AST, alongside the original program, with no distinction between debug code and user code that might inhibit optimisation.

Local Variable Watchpoints

Breakpoints on the modification of local variables, as well as the conditional version of the same, are implemented almost exactly as are line breakpoints. A local breakpoint wrapper is inserted at each local assignment node, and the debugging action happens *after* the child has executed, i. e., when the local holds the newly assigned value.

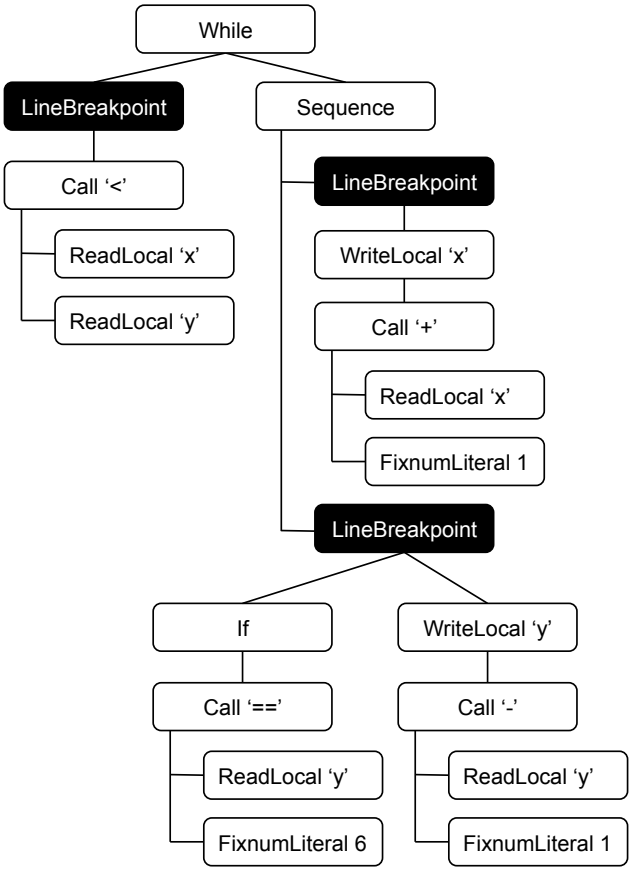


Figure 6.3: AST of Listing 6.3 with a line breakpoint with condition `y == 6`

Profiling

Other researchers have explored the use of wrapper nodes for profiling [91].

6.5 Implementation

This section describes the properties of the underlying Truffle/Graal platform that makes this approach to debugging effective.

6.5.1 The Truffle Compilation Model

The Truffle-based implementation of Ruby is expressed as an AST interpreter [115]. Unlike all other modern implementations of Ruby, we do not generate bytecode and do not explicitly generate machine code. Instead, when running on a JVM with the Graal compiler, Truffle will profile AST execution. When it discovers a frequently executed tree, it takes the compiler intermediate representation of all the methods involved in executing the AST—primarily, all the `execute` methods on the AST nodes—and inlines them into a single method. The powerful intra-method optimisations that the JVM normally applies within methods are applied across all the methods, and Truffle produces a single machine code function for the AST. In our case this is a single machine code function for a single Ruby method. This by-default inlining of AST interpreter methods removes the overhead introduced by inactive wrappers.

6.5.2 Overview

Figure 7.1 summarizes the transitions of an AST with debug nodes under the Truffle compilation model.

The AST is generated from source code by the parser using conventional techniques. In the prototype implementation evaluated in this thesis, the debug wrapper nodes, illustrated as a double circle, were inserted by the parser at each statement. As wrapper nodes are added, a mapping between file names and line numbers to the wrapper node inserted at that location, if any, is built up. A more sophisticated implementation would allow wrapper nodes to be inserted only as they are needed, which would reduce memory consumption.

After enough executions of this AST to trigger compilation, a single machine

code method is produced from all nodes in the AST, including the inactive wrapper node.

When the user wishes to install a line breakpoint, the corresponding wrapper node is found by looking up the source location in the mapping which was built as the wrapper nodes were added. Each wrapper node maintains an `Assumption` object (explained further in Section 6.5.3), which is then invalidated to flag the wrapper node as needing to be replaced with an active wrapper node because a debug action is needed at that location. Invalidating the assumption also triggers deoptimisation (again, this is explained further later on), and the program continues to run in the interpreter, rather than in the compiled code. In the interpreter, inactive wrapper nodes when executed check if they have been flagged to be replaced with active wrapper nodes, because a debug action is needed at that location. When a flagged inactive wrapper node is executed it will specialise, or replace itself, with an active wrapper node with the debug action attached.

Execution continues, and after a period to allow the AST to re-stabilize (for example we have to determine the type of operations and fill inline caches in the modified AST) we again reach the threshold of executions for the AST to be compiled. Graal caches parts of the compilation of the AST so compilation with the replaced node does not have to take as long [99].

With the active wrapper node in the place, the debug action is now run each time that line of the program is executed.

The active wrapper node can be replaced with an inactive node if the debug action is removed, using exactly the same procedure. This cycle can continue repeatedly as the program runs.

Figure 6.5 illustrates how a simplified inactive debug node is compiled to leave zero overhead. The semantic action method of the node, `execute`, checks the assumption that the node should still be inactive. If the assumption is no longer valid the node is replaced. Under compilation, the check is constant and valid. The compiler sees that no exception is thrown and removes the entire `catch` block. The only remaining action is to directly execute the child of the wrapper (the node that is being wrapped). Truffle inlines by default, so there is no method call overhead to execute the child. If the assumption was no longer valid, for example because the user is installing a breakpoint, execution will transition to the interpreter. There the check on the assumption is always performed. Then the exception will be thrown and caught, a new active node will be created and

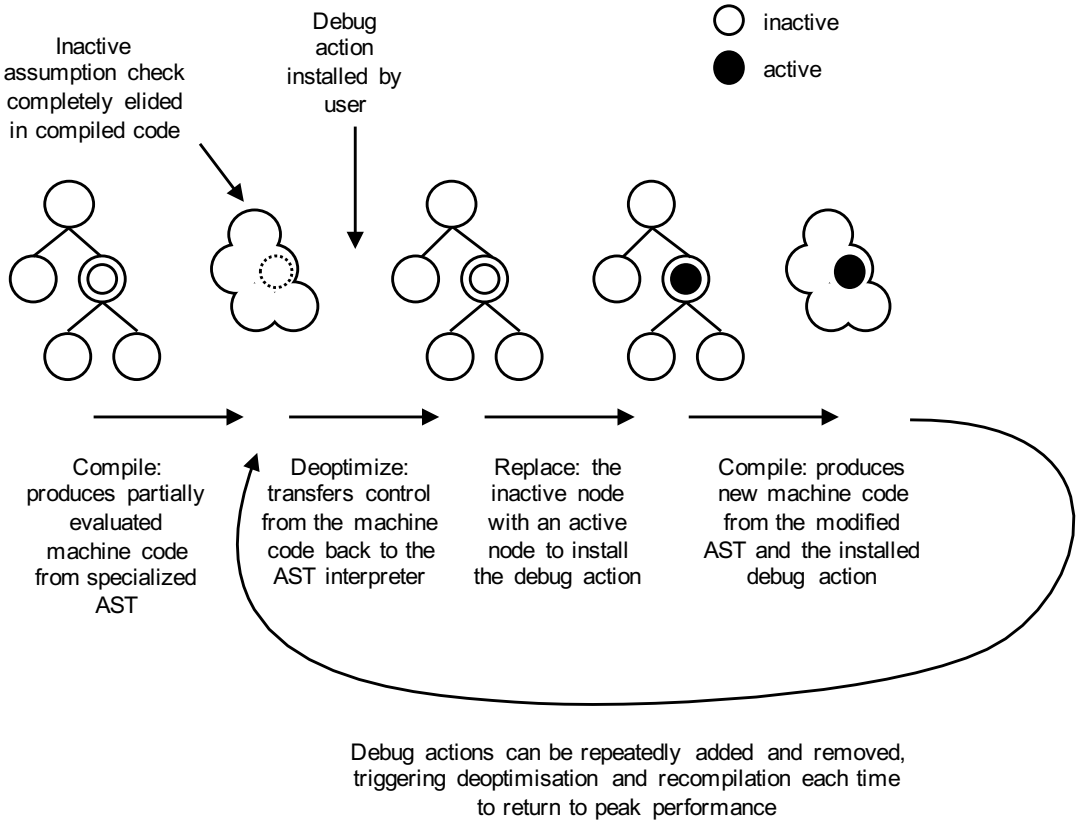


Figure 6.4: Overview of the Truffle compilation model as it applies to debug nodes

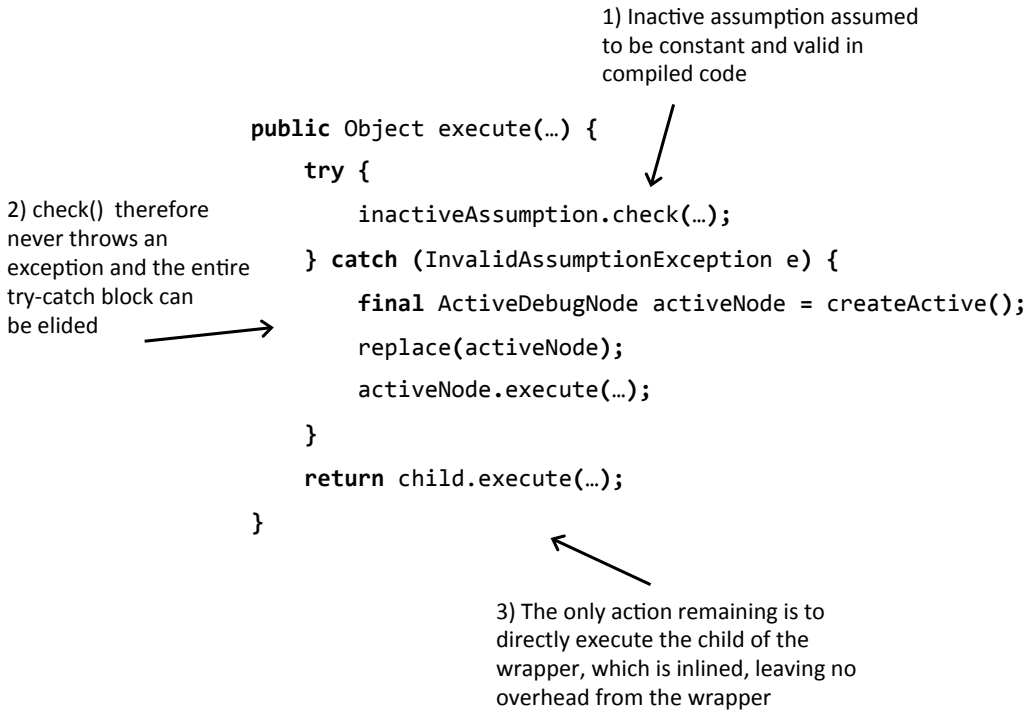


Figure 6.5: Explanation of how code in an inactive debug node (simplified) is compiled to leave zero-overhead

will replace this current inactive node. Execution will then continue with the new active node. At some point Truffle will decide to compile the method again, with the new node in place.

Inactive wrapper nodes play several important roles in ASTs being debugged, even though they compile to nothing most of the time. They make it possible to map significant AST locations, for example the beginning of lines, that could otherwise be reached only by tree navigation. They can be relied upon to persist, even when nodes around them are replaced during Truffle AST optimisation. Finally, they can be activated and deactivated by *self-replacement*, which is Truffle's fundamental (and safe) mechanism for runtime AST modification.

6.5.3 Deoptimisation

Truffle provides two implementations of the `Assumption` class. When Graal is unavailable or the method is being interpreted, `Assumption` is implemented with a Boolean flag and explicit checks as described in section 6.4.3. Since `Assumption` objects are checked often but invalidated rarely, a strategy that treats them as

constant and valid during compilation, but ensures that invalidation of the assumption occurs correctly, can bring large performance benefits. The Graal implementation of `Assumption` on top of the HotSpot JVM provides the mechanism to do this efficiently.

OpenJDK JIT compilers such as *server* [82] and *client* [64] emit machine code at runtime after sufficient invocations of a method, and will then call the machine code version of the method instead of interpreting it. The transition from the initial interpreter into this compiled machine code does not have to be one-way. The transition in the other direction, from machine code to interpreter is called *dynamic deoptimisation* [55]. Part of the complexity of deoptimisation is that invalidated machine code may be already running and on the stack, potentially with more than one activation, and potentially on more than one thread. Multiple activations in a single thread are deoptimised by examining the entire stack when deoptimising and transitioning all activations of affected methods. Activations in other threads are deoptimised by cooperatively halting them at a *safepoint* where threads test a page that has its permissions changed to cause a `segfault` and stop the thread. Safepoints are already emitted by the JVM to support systems such as the garbage collector, so they add no overhead in our system to support debugging multi-threaded applications.

The efficient implementation of `Assumption` is made possible by this mechanism, which Graal exploits by maintaining a list for each `Assumption` object of all machine code that depends on the `Assumption` being valid. The invalidate method on an `Assumption` object instructs the underlying JVM to invalidate all dependent machine code, which triggers OpenJDK JVM deoptimisation.

The Graal VM is specifically designed to enable aggressive speculative optimisations [28]. It uses dynamic deoptimisation internally, and so the mechanism for using it (including *safepoints*, which are discussed in depth in Chapter 7) is already present in all compiled code. We can therefore reuse that mechanism for debugging with no additional runtime overhead.

6.5.4 Repeated Invalidations of Assumptions

When a debug node is initially added to an AST that is already compiled, deoptimised is triggered for the first time by invalidating the `Assumption` object associated with the AST. As shown in Figure 7.1, the AST can be recompiled with the debug node in place. `Assumption` objects can only be invalidated once,

and when invalidated they stay in the invalidated state. To support recompilation, the Truffle API was modified to add a class `CyclicAssumption`. As shown in Listing 6.5, the `CyclicAssumption` holds a mutable reference to a current `Assumption` object. When this object is invalidated a new one is automatically created. Assumptions which have been invalidated and are no longer referenced are collected by the garbage collector as any object, so as many can be created as are needed as the program runs, and the cycle of optimisation and deoptimisation is not bounded to a limited number.

```
1  class CyclicAssumption {
2      private Assumption currentAssumption;
3
4      public void invalidate() {
5          Assumption oldAssumption = currentAssumption;
6          currentAssumption = Truffle.getRuntime().createAssumption();
7          oldAssumption.invalidate();
8      }
9
10     public Assumption getAssumption() {
11         return currentAssumption;
12     }
13 }
```

Listing 6.5: Implementation of `CyclicAssumption` (simplified)

6.5.5 Expectation

The inlining of AST interpreter methods and the use of dynamic deoptimisation via `Assumption` objects instead of explicit checks to enable debug operations means that our debugger implementation will have no peak temporal performance overhead at all when debugging is enabled but not in use. After optimisation, a tree with an inactive line breakpoint wrapper becomes no different in terms of the JVM JIT compiler IR than if the wrapper had not been added, as it had no body after the assumption check was removed, and we remove method boundaries.

It is also possible to inline a trace method or breakpoint condition, rather than making a method call. A copy of the trace method's AST can be inserted as a child node of the active wrapper. In this position it is optimised in exactly the same way as user code, as if inserted at that point in the source code. Part

of the reason that `set_trace_func` is expensive in existing implementations (see section 8.10) is that the trace method is passed the current environment as a parameter. When our debugger inlines a trace method, Graal observes through escape analysis [63] that the environment argument can be optimised.

However, to implement `set_trace_func` the developers of Topaz needed to define the trace method as a green variable in their main JIT object and at each *merge point* where the interpreter could enter a compiled trace. We believe that our system where an `Assumption` is only of concern to the subsystem that is using it, rather than a 'global' object, is more elegant.

6.6 Evaluation

We evaluated the performance of our implementation of `set_trace_func` and debugging JRuby+Truffle against other implementations of Ruby interpreters and debuggers, using the systems and techniques described in Chapter 4.

In this chapter we are primarily interested in the *overhead*. This is the *cost* of using a feature in terms of how much performance is reduced by the feature. We consider the overhead of each feature in each implementation in four configurations. The *disabled* metric is our baseline. It is the performance of the system when the feature has been disabled, either through normal configuration, or through modification of the implementation. The *before* metric is the performance of the system when the feature is enabled, but when it is not actually in use, and before it has ever been in use in that process. The *during* metric is the performance of the system when the feature has been enabled and is actually in use. The *after* metric is the performance of the system after we have stopped using the feature.

In an ideal case we would hope there there is no difference between *disabled* and *before*, which would mean that having that feature available does not have an impact on performance. There would be no, or reasonable and minimum, difference between *before* and *during*, which would mean that actually using the feature does not reduce the performance of the system. It is less of an obvious requirement, but we would also like to have no difference between *before* and *after*, meaning that the system is able to return to peak performance after the feature is no longer being used and the process is no longer being debugged.

In our results we will show the absolute time for *disabled*, with standard

deviation and standard error relative to the mean as a percentage. We then show the performance of *before*, *during* and *after* relative to the performance of *disabled* to make the overhead clear. Lower is therefore better, and a value of zero means that there is no overhead.

We evaluated each configuration at its peak performance, so for each configuration it is after a period of warmup after that particular configuration has been reached. We did not evaluate the performance of a change in state, such as hitting a breakpoint or entering the interactive debugging, as we consider these to be offline operations and not relevant to our idea of peak performance.

6.6.1 Versions

We compared against version 2.1.0 of MRI and stdlib-debug, version 2.2.4 of Rubinius, a development build of JRuby at revision 59185437ae86, a development build of Topaz at revision 4cdaa84fb99c, built with RPython at revision 8d9c30585d33.

We used ruby-debug at version 1.6.5, and jruby-debug at version 0.10.4. jruby-debug is not compatible with the development version of JRuby, so we ran experiments using jruby-debug with the latest compatible version, JRuby 1.7.10.

The version of JRuby+Truffle evaluated for this chapter was as at revision 59185437ae86. The Graal dynamic compiler was at version 0.1 for this revision. The implementation of `set_trace_func` and debugging in later revisions of JRuby+Truffle has been modified and expanded considerably, and has now become part of the Truffle framework itself.

6.6.2 Benchmarks

For this chapter we only evaluated against two synthetic benchmarks, *fannkuch* and *mandelbrot*. The benchmarks in this chapter are only used as a base on which to install a `set_trace_func` `Proc` and debugger breakpoints, so the simplicity of these benchmarks is not important.

6.6.3 Overhead of `set_trace_func`

We evaluated the overhead of enabling and using `set_trace_func`.

MRI and Topaz do not have an option to allow us to disable `set_trace_func`,

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI	0.995 (0.006) ±0.142%	0.1x	24.6x	0.1x
JRuby	0.358 (0.008) ±0.514%	3.9x	199.4x	3.7x
Topaz	0.154 (0.001) ±0.204%	0.0x	661.1x	0.0x
JRuby+Truffle	0.091 (0.003) ±0.692%	0.0x	4.0x	0.0x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI	2.015 (0.001) ±0.014%	0.0x	30.7x	0.0x
JRuby	0.992 (0.013) ±0.304%	2.8x	153.5x	2.8x
Topaz	0.073 (0.000) ±0.054%	0.2x	5680.4x	0.0x
JRuby+Truffle	0.060 (0.000) ±0.179%	0.0x	5.0x	0.0x

Table 6.1: Overhead of `set_trace_func`

so were patched to remove the functionality. For JRuby we had ran in the default configuration for *disabled*, and with the `--debug` flag in order to be able to use `set_trace_func`. For JRuby+Truffle we added an option to not create the wrapper nodes, for the *disabled* configuration. Rubinius does not support `set_trace_func` at all, so is not evaluated in this subsection.

Table 6.1 shows the overhead of `set_trace_func` in different implementations and configurations.

MRI shows low overhead because one extra memory read per line is a tiny proportion of the work done in executing the rest of the line under their execution model. The overhead when a trace method is installed is high but not unreasonable given it has no mechanism to elide the allocation of a `Binding` object and so must actually allocate it on the heap for each trace event.

JRuby's initial overhead results from having to disable compilation to JVM bytecode, which is required in order to use the feature. The overhead of calling the trace method is limited by having to allocate the `Binding` object.

Topaz has low but statistically significant overhead for enabling tracing. However the implementation does not appear to be optimised for having a trace method actually installed, showing a pathological overhead as large as three orders of magnitude.

JRuby+Truffle shows very low overhead for enabling tracing, and a reasonable overhead of 4–5x to install a trace method. Our implementation inlines the trace

method, allowing the binding (the object representing the current environment) to be elided if not actually used. If the trace method is used, and if escape analysis determines that the binding cannot be referenced outside the method, then the frame can be allocated on the stack for better performance than via default heap allocation.

6.6.4 Overhead of a Breakpoint on a Line Never Taken

We evaluated the overhead of using the various debuggers to set a breakpoint on a line in a method which is on the critical path, but where the actual line is not taken. This represents the cost of setting a line breakpoint on some rarely taken erroneous path where the programmer wants to enter the debugger to diagnose the problem. The breakpoint was set on a line in the inner loop of the benchmarks. The condition we used to guard the line was not statically determinable to be always false by any of the implementations, so that the condition would not be optimised away.

The question we are asking is this: *if such an erroneous state is only observed intermittently, such as once a week, what is the cost of having the breakpoint set during the whole run of the program to catch the one time when it is?*

For this experiment, we considered multiple combinations of implementation and debugger where possible. For example, we show JRuby with both stdlib-debug and jruby-debug. In later summaries, we show only the best performing combination. Normally, JRuby+Truffle would detect that the branch is never taken during interpretation and speculatively elide it for compilation, but we disabled conditional branch profiling for all these experiments.

Table 6.2 shows the performance of different implementations and debuggers. The overhead of using stdlib-debug in either MRI or JRuby is extremely high as it is based on the already inefficient implementations of `set_trace_func`. The native extension variants ruby-debug and jruby-debug show two orders of magnitude less overhead, bringing it down to around a reasonable 5x. Rubinius also has a reasonable overhead of 1.2–6.8x. JRuby+Truffle shows very low overhead for all states. Overhead is negative in some cases due to normal error in sampling.

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI/stdlib-debug	1.043 (0.006) ±0.124%	154.2x	182.9x	196.3x
MRI/ruby-debug	1.043 (0.006) ±0.124%	4.3x	4.7x	4.3x
Rubinius	1.459 (0.011) ±0.174%	4.5x	6.8x	3.6x
JRuby/stdlib-debug	0.562 (0.010) ±0.402%	1375.2x	1609.2x	1573.3x
JRuby/jruby-debug	0.562 (0.010) ±0.402%	4.5x	43.3x	41.9x
JRuby+Truffle	0.091 (0.003) ±0.692%	0.0x	0.0x	0.0x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI/stdlib-debug	2.046 (0.001) ±0.009%	139.6x	179.0x	166.8x
MRI/ruby-debug	2.046 (0.001) ±0.009%	5.5x	5.6x	5.5x
Rubinius	1.151 (0.002) ±0.031%	4.6x	11.7x	3.8x
JRuby/stdlib-debug	1.096 (0.008) ±0.170%	1698.3x	1971.4x	1884.1x
JRuby/jruby-debug	1.096 (0.008) ±0.170%	4.6x	49.8x	48.3x
JRuby+Truffle	0.060 (0.000) ±0.179%	0.0x	0.0x	0.0x

Table 6.2: Overhead of setting a breakpoint on a line never taken (lower is better)

6.6.5 Overhead of a Breakpoint With a Constant Condition

Finally, we evaluated the overhead of setting a line breakpoint with a constant condition that is statically determinable to always evaluate to `false`. This tests the overhead of a conditional breakpoint where the condition itself should have no overhead. Again the breakpoint was set on a line in the inner loop of the benchmarks.

We could not find any support in stdlib-debug for conditional breakpoints, so it is not evaluated in this subsection.

Table 6.3 shows that results for MRI, Rubinius and JRuby are broadly the same as before, except with a significant additional overhead caused by evaluating the condition. JRuby+Truffle now shows a significant overhead when the conditional breakpoint is installed. Although the condition is constant, we were not yet able to inline the condition in the line where the breakpoint is installed, so this overhead represents a call to the condition method.

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	1.043 (0.006) ±0.124%	4.3x	25.8x	4.2x
Rubinius	1.459 (0.011) ±0.174%	3.7x	187.4x	3.7x
JRuby/jruby-debug	0.562 (0.010) ±0.402%	4.6x	41.2x	41.4x
JRuby+Truffle	0.107 (0.003) ±0.528%	0.0x	1.7x	0.0x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	2.046 (0.001) ±0.009%	5.5x	35.4x	5.5x
Rubinius	1.151 (0.002) ±0.031%	4.6x	662.8x	4.0x
JRuby/jruby-debug	1.096 (0.008) ±0.170%	4.3x	48.0x	47.4x
JRuby+Truffle	0.059 (0.001) ±0.188%	0.0x	8.1x	0.0x

Table 6.3: Overhead of setting breakpoint with a constant condition (lower is better)

6.6.6 Overhead of a Breakpoint With a Simple Condition

Finally, we looked at the overhead of setting a line breakpoint with a simple condition, comparing a local variable against a value it never holds. This tests the normal use of conditional breakpoints, such as breaking when some invariant fails. Again the breakpoint was set on a line in the inner loop of the benchmarks.

Table 6.4 shows that the overhead when there is a simple condition to test compared to a constant condition is not great in MRI. The overhead in JRuby+Truffle when the breakpoint is installed is increased but is still reasonable at up to 10x.

6.6.7 Summary

Table 6.5 summarizes the results across both benchmarks, using the highest performing debugger implementation for each implementation of Ruby. We show the overhead for different debug tasks, in each case compared to when `set_trace_func` or debugging is disabled. Figure 6.6 shows self-relative performance on a logarithmic scale, with relative performance of 1 being no-overhead and one vertical grid line being an extra order of magnitude of overhead.

JRuby+Truffle has on average 0.0x overhead for enabling tracing, debugging and setting a breakpoint on a line never reached. For constant and simple conditional breakpoints on lines in the inner loop of the benchmarks which we cannot

	Fannkuch			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	1.043 (0.006) ±0.124%	4.3x	31.6x	4.2x
Rubinius	1.459 (0.011) ±0.174%	3.7x	187.7x	4.5x
JRuby/jruby-debug	0.562 (0.010) ±0.402%	4.4x	86.2x	41.9x
JRuby+Truffle	0.107 (0.003) ±0.528%	0.1x	10.1x	0.1x
	Mandelbrot			
	Disabled (s (sd) se)	Before	During	After
MRI/ruby-debug	2.046 (0.001) ±0.009%	5.6x	50.7x	6.7x
Rubinius	1.151 (0.002) ±0.031%	4.7x	659.8x	4.5x
JRuby/jruby-debug	1.096 (0.008) ±0.170%	4.5x	105.3x	46.8x
JRuby+Truffle	0.059 (0.001) ±0.188%	0.0x	7.9x	0.0x

Table 6.4: Overhead of setting breakpoint with a simple condition (lower is better)

	MRI	Rubinius	JRuby	Topaz	JRuby+Truffle
Enabling set_trace_func	0.0x	n/a	2.3x	0.1x	0.0x
Using set_trace_func	26.6x	n/a	39.9x	2714.2x	4.5x
Enabling debugging	4.9x	4.6x	4.6x	n/a	0.0x
Breakpoint on a line never taken	5.1x	9.3x	46.5x	n/a	0.0x
Breakpoint with constant condition	30.6x	425.1x	44.6x	n/a	4.9x
Breakpoint with simple condition	41.2x	423.7x	95.8x	n/a	9.0x

Table 6.5: Summary of overheads (lower is better)

optimise away entirely JRuby+Truffle has a very reasonable overhead in the range 5–9x: about an order of magnitude less than other implementations.

When interpreting these results we should also keep in mind the extreme performance variation among language implementations which we demonstrated in Subsection 3.3.6. These overheads are *on top* of those differences. In terms of absolute wall-clock performance, JRuby+Truffle is over two orders of magnitude faster than the next fastest debugger, MRI with ruby-debug, when running with a breakpoint on a line never taken.

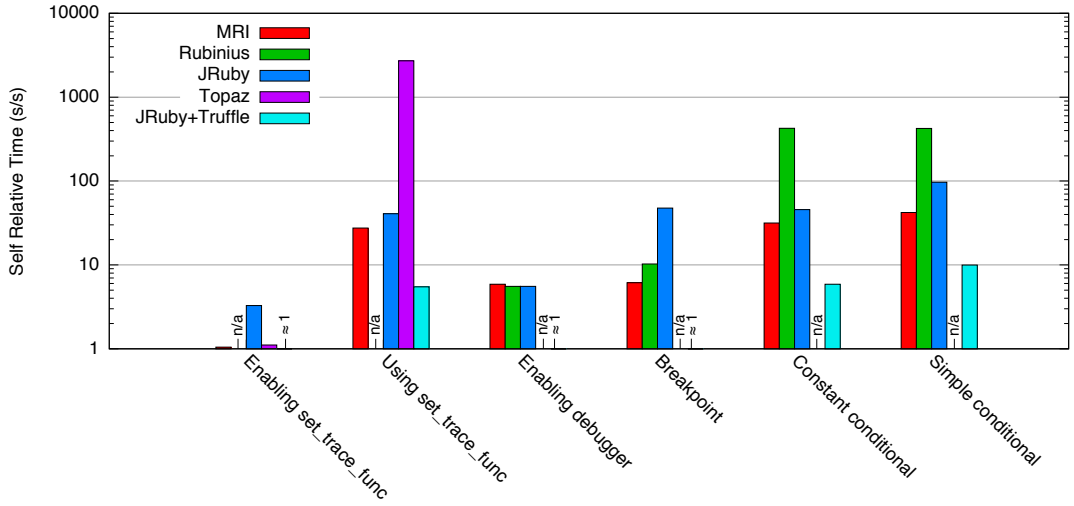


Figure 6.6: Summary of relative performance when using debug functionality (taller is exponentially worse)

6.7 Related Work

6.7.1 The Self Programming Language

Debugging support was one of the primary motivations behind the development of *dynamic deoptimisation* in the Self programming language, which was claimed to be "the first practical system providing full expected [debugging] behavior with globally optimised code" [55]. Along with other seminal innovations in Self, this derived from its creators' firm commitment that the experience of using a language is fully as important as performance [106]. Research on the implementation of Self produced many of the techniques applied and advanced in this thesis, such as deoptimisation and polymorphic inline caching.

Debugging in the original Self was primitive; setting a breakpoint required manually inserting "a send of `halt` into the source method". It was also deeply entwined with the language implementation. For example, the `finish` operation was implemented by "changing the return address of the selected activation's stack frame to a special routine ...".

Two decades of progress in the underlying technologies have led to the Truffle platform, which supports multiple languages, and into which nearly transparent debugging code can be inserted without the language specificity and fragility of its original incarnation.

6.7.2 Debugging optimised Code

Debugging statically compiled, optimised code has been a problem worthy of many PhD dissertations. As the Self creators pointed out, that work generally gave priority to optimisation, and results generally were complex and supported only limited debugging functionality [55]. The Ruby debugger demonstrates that the compromise can be avoided.

6.7.3 Wrapper Nodes

The idea of wrapping nodes to transparently introduce extra functionality was applied before in a machine model for aspect-oriented programming languages [52]. The abstractions used there are generic enough to be used for debugging as well, as this work shows.

6.8 Summary

Early experience with an experimental Ruby debugger suggests that it is possible to build debuggers on the Truffle platform without the compromises listed in the introduction.

- *Performance*: Runtime overhead is extremely low, and is arguably minimal relative to optimisation supported by Truffle. Inactive AST node wrappers incur zero overhead when dynamically optimised along with program code. Activated debugging actions, whether expressed in Java or the implemented language, are subject to full optimisation.

- *Functionality*: We have yet to see any limitations imposed by Truffle on the kind of debugging functionality represented in the prototype.

- *Complexity*: There is almost no interaction between the inserted debugging code and Truffle's mechanisms for compilation, dynamic optimisation, and dynamic deoptimisation. Debugging code needs to only follow standard Truffle techniques for AST construction and use the `Assumption` class correctly. The "wrapper" AST nodes that implement debugging actions are almost completely transparent to the flow of program execution around them.

- *Inconvenience*: We see no reason that such a debugging infrastructure should not be present in any environment, developmental or production.

This approach is applicable to the range of languages that can be implemented on Truffle. Nothing reported here other than the `set_trace_func` functionality is specific to Ruby.

Moreover, this approach places only modest demands on other parts of a language implementation. We anticipate language implementers adding debugging support incrementally during development, with evident advantage to both themselves and early users. We also anticipate applying this general approach to supporting development tools other than debugging.

As in Chapter 5, the techniques introduced in this chapter were shown to have zero-overhead in run-time compared to the case without the language feature that uses them, when the program is stable and has been optimised as described in Chapter 4. There will however be overheads in compilation time and memory consumed by the compiler, but we were not attempting to reduce or maintain these other metrics at previous levels.

Chapter 7

Safepoints in Dynamic Language Implementation

7.1 Introduction

As with debugging and described in the previous chapter, optimisation can be in conflict with some programming language features. In Ruby, some features of the language seem to have evolved because their implementation was trivial given the architecture of MRI at the time. Implementations of Ruby which want to achieve higher performance using optimisations have found that the techniques they want to use and the optimisations that they want to apply have made these features problematic.

We would like to be able to support these language features without reducing performance. As in the previous chapter, the ideal situation is for there to be zero-overhead for providing the features compared to if they were disabled. Other goals are to limit the performance impact when they are actually used.

An example of one of these features is the `ObjectSpace` module. This provides access to the object graph of the running program, such as being able to iterate through all live objects of the program and produce a count, as shown in Listing 7.1.

In MRI the implementation of `each_object` is trivial because as a VM implemented from scratch in C they had to implement their own memory allocator and garbage collector. To implement `each_object` all they had to do was walk their own heap data structure in the same way as the garbage collector's sweep phase would. Indeed, in MRI, `each_object` is implemented in the `gc.c` file as a

```ruby
n = 0
ObjectSpace.each_object do |object|
  n += 1
end
puts "there are #{n} live objects"
```

Listing 7.1: An example use of Ruby's `ObjectSpace.each_object` method

simple extension of the collector.

We can give a concrete example of an optimisation in conflict with this language feature, *allocation removal*. If allocations are removed and objects either allocated on the stack, or never allocated at all, then the implementation technique used in MRI does not work. The technique also does not work if the implementation is reusing an existing garbage collector that does not provide a similar language feature, which few high performance systems will do due to the conflict with optimisations.

JRuby does not support `each_object` as the JVM does not allow running programs to introspect the heap. If the `-Xobjectspace.enabled=true` option is used, all objects will be added to a *weak map* (a map where the references to the keys and objects is ignored by the garbage collector's mark phase) when they are allocated. As well as this option turning off compilation, limiting any serious performance gains, adding all objects to a global data structure will cause every allocation to escape. Topaz uses RPython's sophisticated garbage collector which does itself allow object enumeration, but this limits `each_object` in Topaz to only work with objects that are managed by the garbage collector, which does not include objects that have had their allocation removed. This makes the Topaz implementation unsound[1] as the programmer does not know which objects are not managed by the GC. Worse than that, this is an example of where an optimisation can change the behaviour of the program. Rubinius implements `each_object` using the same technique as MRI, but does not perform any escape analysis or allocation removal and therefore has lower performance anyway.

Note that a positive result from escape analysis and subsequent allocation removal does not demonstrate that an object is not long-lived or important to the application. In fact, that an object is not reachable globally and so does not escape is actually a reason why the programmer could be trying to reach it via

[1] https://github.com/topazproject/topaz/issues/843

`each_object`.

To solve this problem for JRuby+Truffle we decided instead of using the GC we would manually walk the object graph. Objects which are referenced by the VM are called *root objects*. This includes the main object, thread objects, global variable values, and a few other similar objects. The set of root objects also includes objects referenced on the stack, such as through local variables. To walk the object graph to find all live objects, we need to have the complete set of root objects. Finding objects such as those in global variables is easy, but accessing all current stack frames on all threads is not possible on a conventional JVM. Truffle and Graal provide functionality to introspect the stack of the current threads, and will automatically *reify* (actually allocate) objects that had their allocation previously removed. This only leaves us with the problem that stacks can only be introspected as an action of the corresponding thread, and that we need to stop the object graph being mutated as we read it.

Our solution and key contribution of this chapter is a technique for languages implemented on top of the JVM to cause all managed threads to pause in a consistent state that we call a *guest language safepoint* and to perform some action that is dictated by the thread which triggered the safepoint. For `each_object` this action is to introspect their stacks and continue to explore the object graph to build the list of live objects that is needed.

We use the term *safepoint* because the technique is similar to, and is implemented using VM safepoints which are already present for existing VM services such as GC. Our implementation is zero-overhead in that peak performance when enabled compared to when disabled is the same. In this chapter we take our evaluation further than we did in Chapter 6 and we show that the generated machine code for these two configurations is the same, and we also evaluate the cost of the transition from optimised to unoptimised and back that using the feature causes.

The research contributions in this chapter were developed independently, initially to support the `ObjectSpace.each_object` language functionality (described later). Generalisation and application for other functionality such as examining call stacks, attaching a debugger and inter-thread communication (all described later) were also independent contributions. The implementation was improved and correctness improved in collaboration with Benoit Daloze at Johannes Kepler Universität, Linz, and the work was evaluated and written up in collaboration with Benoit Daloze and Daniele Bonetta at Oracle Labs [23].

7.2 Safepoints

Safepoints are probably most commonly thought of as a technique with which to implement many garbage collection algorithms in multi-threaded VMs. Although there are fully concurrent garbage collectors such as C4 [104], most tracing garbage collectors at some point will need to *stop-the-world*, or pause all threads, so that some operations can be performed on the heap without it being concurrently modified. The term *safepoint* is used because another requirement is that the threads stop in a state where the runtime is able to examine them.

In a VM with no concurrency this isn't needed as a single thread can simply decide to pause itself. Likewise in systems where there is a global interpreter lock [79] such as MRI each lock exchange allows the same goal to be achieved.

The term *safepoint* itself is overloaded and refers to several overlapping concepts, so we will disambiguate by qualifying it. A *safepoint action* is some computation we want to run. A *VM safepoint* is a state where all VM threads are either running a safepoint action, or waiting for another thread to finish doing so. Our contribution, a *guest-language safepoint*, is the same thing, but implemented in the guest language implementation, rather than in the underlying VM. The points within a program where any of these safepoints can be entered is called a *safepoint check*, or some literature uses the term *yieldpoint*.

The key problem in safepoints is the implementation of the safepoint check. We need the ability for one thread to be able to send a message to all others. Those other threads may be deep inside tight computation loops that are potentially endless, or they may be waiting inside a blocking system call, or waiting on a lock.

7.2.1 Techniques

Implementation techniques for safepoint checks are surveyed by Yi Lin et al [70].

The simplest technique is *condition polling*. Here, there is a global variable that acts as a flag. All threads periodically read this flag in a branch with logic to pause or perform any other action if the flag is set. To pause all other threads, a thread can set the variable and then use some other logic to communicate with them to check they have entered the safepoint and to send the action that should be run. The advantage of this technique is that it is very simple to implement.

The downsides of condition polling are that a volatile read is expensive, a register must be used to read the flag into, two instructions are required in most instruction sets for the read and then the conditional branch, and the processor is relied upon to successfully predict the branch each time to avoid pipeline stalling. This is the technique used in Rubinius. In fact, Rubinius has multiple flags for different applications of safepoints that it must check regularly.

A more advanced technique used by most serious implementations of JVMs such as HotSpot is *trap-based polling*. This technique improves on condition polling by removing the branch instruction, and instead uses the processor's memory protection system. To trigger the safepoint, the permissions on the page where the flag is stored are changed so that the thread cannot read the variable. The next time the safepoint check runs the read will cause a protection fault and a signal will be raised. The process can handle this signal by jumping to logic for the safepoint, as before. The advantages of this approach are that we have reduced the instructions needed from two to one, and as the value read is no longer actually used, just the side effects of reading it, we can use an instruction such as `test` on the x86 architecture which sets flags but does not clobber a register. We are also not dependent on the branch predictor as there is no longer a branch. The disadvantage of this approach is that it requires signal handling that may be difficult to get right.

Another advanced technique is *code patching*. Here the generated code includes no instructions for the safepoint except for a sequence of `noop` instructions. To trigger the safepoint these `noop` instructions are replaced by code that will jump to the safepoint handling logic, such as with the explicit condition polling described above. The advantage of this technique is that it may be the lowest overhead of all [70], but the disadvantage is that it may be difficult to maintain a list of all the locations in machine code that need to be patched and to update them all quickly.

7.2.2 Safepoint Check Scheduling

Whichever way the safepoints are implemented, there needs to be a decision about how to *schedule* them, or how often to insert them into the program. Safepoint checks need to be scheduled often enough to meet two conditions. First, the latency between requesting a safepoint and a check being executed to respond to it needs to be low enough for the particular applications. A GC might need a

very low latency to limit pause times (anecdotally some VMs spend more time waiting for all threads to find a safepoint than they do doing the collection itself), but a user interaction such as attaching a debugger may be content with higher latency. Second, the latency must be bounded. High latency, or variable latency, may be tolerable, but infinite latency is not. It must not be possible for a thread to spin within a tight inner loop and never perform a safepoint check.

In general the accepted solution is to check for a safepoint request at least once per method (or other unit of generated machine code such as a trace), and at least once in inner-most loops. Optimisation phases may remove safepoint checks from loops if they can be proved to not run long enough to cause significant latency. For example, Graal will remove safepoints from inner-most loops with a statically countable number of iterations, or with a counter that is of width less than or equal to 32 bits, as such a loop will never run for very long (only so much code can be executed dynamically without any kind of looping or method call). We've disabled this optimisation in Graal for examples of machine code shown below, as otherwise it can be hard to provoke Graal to include a safepoint in code that is simple.

7.3 Guest-Language Safepoints

The JVM internally uses safepoints for several applications, such as GC, debugging and invalidating speculative optimisations, but these are all services that are transparent to the running program. We would like to make safepoints available as an API so that we can use it in our implementation of Ruby, in order to implement Ruby language features such as `each_object`. We call these *guest-language safepoints*.

In Listing 7.2 we sketch what this Java API could look like.
`pauseAllThreadsAndExecute` is a static method that accepts a lambda (or perhaps an instance of an anonymous instance of `Runnable`) that is the safepoint action. The method causes all threads to enter a guest-language safepoint, and when they have done so it runs the action concurrently on each thread. When all threads have completed the action, all can leave their safepoint and continue running the program as normal. As safepoints are an exception to the normal runtime condition and may not be reentrant or a safe place to run arbitrary user code, we also have a method `runAfter` which also accepts an action, but

will run it after all threads have left their safepoint and are running under normal conditions. Normal useage of `runAfter` is to use it within the action of `pauseAllThreadsAndExecute` to defer an action.

```
Safepoints.pauseAllThreadsAndExecute(() -> {
  // logic to execute in the safepoint here
});

Safepoints.runAfter(() -> {
  // logic to execute after the safepoint here
});

// Call at least once per method and once per loop iteration
Safepoints.poll();
```

Listing 7.2: Sketch of an API for safepoints

A final method, `poll` is inserted within the guest language implementation with the same scheduling as we described earlier: once per method and once per loop. Listing 7.3 shows example polls similar to those in the JRuby+Truffle source code. As we will show later on, the scheduling needs to be at least sufficient but does not need to be manually optimised beyond that.

```
// Poll at method root node execute method

public Object execute(VirtualFrame frame) {
  Safepoints.poll();
  return body.execute(frame);
}

// Poll at the head of a loop

while (...) {
  Safepoints.poll();
  body.execute(frame);
}
```

Listing 7.3: Example locations for a call to `poll()`

7.4 Applications of Safepoints in Ruby

7.4.1 Enumerating Live Objects

In Section 7.1 we described one application of guest-language safepoints in JRuby+Truffle, `each_object`.

The case with a single sequential thread can be implemented using existing Truffle APIs that allow introspection of the guest-language stack. When running on a standard JVM the guest-language stack is represented as a chain of heap-allocated method activation objects. When dynamically optimising using the Graal compiler, dynamic deoptimisation is used to cause reification of all objects which have been subject to allocation removal, including the guest-language stack and any escape-analysed objects.

All Ruby objects found on the stack become root objects, as well as a small set of static roots such as the Ruby main object. From these roots we can find all live objects by recursively visiting all the fields of each object to find further live objects. An object graph may be cyclic, so we maintain a visited set. This is effectively implementing our own sweep phase of a tracing garbage collector, and gives us set of live objects.

This solves the case for a program with a single sequential thread. In a program with multiple threads, or using other concurrent Ruby language features such as finalisers, which run on a dedicated thread, we also need to be able to instruct all other threads, as well as the current one, to stop and introspect their stack to find more roots. Our guest-language safepoints API can be used to do this.

Listing 7.4 shows the allocation of a set to hold live objects from all threads in line 1. Line 2 uses the `Safepoints.pauseAllThreadsAndExecute` call, passing it a lambda for the safepoint action which will be run on all guest-language threads. Running the action, each thread visits its own call stack in line 5. As these actions run concurrently, line 4 synchronizes on the live set.

7.4.2 Intra-Thread Communication

Ruby allows one thread to pre-emptively kill another with the `Thread.kill` method, but the language we are writing our implementation in, Java, does not have this functionality. Even if it did, we may want to run clean-up actions or

```
1 Set<Object> liveObjects;
2 Safepoints.pauseAllThreadsAndExecute(() -> {
3   synchronized (liveObjects) {
4       visitCallStack(liveObjects);
5   }
6 });
```

Listing 7.4: Using guest-language safepoints to implement `each_object`

application `ensure` blocks (the equivalent of Java `finally` blocks) which could run arbitrary code.

We can implement this functionality using our guest-language safepoints. Listing 7.5 shows a variable `targetThread` set on line 1 to the thread we wish to target. This variable is captured in the closure of the action that is run in the safepoint. Each thread compares themselves against the target thread, and if the action applies to them an exception is thrown. This causes the thread to exit after normal unwinding behaviour.

```
1 targetThread = ...
2 Safepoints.pauseAllThreadsAndExecute(() -> {
3   if (currentThread == targetThread) {
4     Safepoints.runAfter(() -> {
5       throw new KillException();
6     }
7   }
8 });
```

Listing 7.5: Using guest-language safepoints to implement `Thread.kill`

Note that we use the `Safepoints.runAfter` method to defer actually throwing the exception until all threads resumed normal running. This is because handling the exception may cause arbitrary Java or Ruby code to run, which of course may mean other routines that need to use safepoints. The guest-language safepoint system is not re-entrant, and it is not clear that making it so would be useful, given that `Safepoints.runAfter` is available.

In this example we have killed the thread by raising an exception, but really we can do anything we want in the lambda passed to `runAfter` after identifying the target thread, and it becomes a general mechanism for sending code from one thread to be executed on another. For example, Ruby has another method

`Thread.raise` that allows an arbitrary exception to be raised in the target thread, which we implement in almost exactly the same way.

7.4.3 Examining Call Stacks

One simple way of finding out what a program is doing is to inspect its threads' call stacks. `jstack` does this for Java programs, but its implementation requires VM support and so it is normally not possible to implement the same functionality for a guest language.

Using our API we implemented a version of `jstack` for JRuby+Truffle. We added a VM service thread that listens on a network socket. Our equivalent of the `jstack` command sends a message to this socket, and the service thread uses our safepoint API to run an action on all threads that prints the current guest-language stack trace.

```
1 Safepoints.pauseAllThreadsAndExecute(() -> {
2   printRubyBacktrace();
3 });
```

Listing 7.6: Safepoint action to print stack backtraces

7.4.4 Debugging

We discussed in Chapter 6 how to implement a zero-overhead debugger for programming language built on a specialising AST interpreter and a compiler with dynamic deoptimisation. We demonstrated that by re-using VM safepoints a debugger can attach and remove breakpoints in a running program with zero overhead until the breakpoints are triggered. In that work the debugger was in-process and could only be used by writing application code to enter the debugger where commands could be entered. Using our guest-language safepoints we can extend that functionality to allow a debugger to be attached remotely. We reused the same VM service thread as before that listens on a network socket. When a message is received the service thread runs a safepoint action on the main thread telling it to enter the debugger, from where breakpoints can be added and removed and the program inspected.

Listing 7.7 shows the action, which is similar to Listing 7.5 except we unconditionally use the main thread to start the debugger. We could also provide

an option to break on a particular thread, or to execute a debug action non-interactively such as installing a breakpoint in order to break on a particular line. Here we have decided to put the logic into the safepoint action itself, and not use a deferred action, as we want all threads to stay paused while the user debugs.

```
Safepoints.pauseAllThreadsAndExecute(() -> {
  if (currentThread == mainThread) {
    enterDebugger();
  }
});
```

Listing 7.7: Safepoint action to enter a debugger

7.4.5 Additional Applications

Additional applications of guest-language safepoints include deterministic parallel execution of JavaScript such as the RiverTrail model [53], where code is optimistically run in parallel and monitored for conflicts. If a conflict is detected the parallel approach is incorrect and all the threads involved need to be aborted. This can be achieved by using a guest-language safepoint [23].

7.5 Implementation

7.5.1 A flag check

The simplest implementation of guest-language safepoints on the JVM would be to use the condition polling technique, described above. We would define a single field to act as a flag for whether threads should enter a safepoint. Each time the `poll()` method is called the flag is read. If it is set to true then we enter the safepoint logic itself. To trigger the safepoint, the value of the flag is simply changed.

The variable we use as a flag needs to be volatile, as the JVM memory model allows non-volatile fields to be read once and the value re-used unless there is a synchronization point. In practice this may mean that a method containing an infinite loop may only actually read the field once, and the check within the loop

could just use a cached value. A thread running the infinite loop would never detect the guest safepoint.

Listing 7.8 illustrates this. The flag variable is declared as volatile on line 1. The implementation of `poll` just has to read the flag on line 5. If it is set, we enter the safepoint logically. Briefly, this needs to wait for all other threads to realise the flag has been set, execute the action for this thread, wait again for all other threads to finish running the action, and then run any deferred actions before continuing. The implementation of `pauseAllThreadsAndExecute` sets the flag to true, and then like `poll()` it runs the same safepoint action, but including resetting the flag. Notice that the thread which triggers the safepoint using `pauseAllThreadsAndExecute` won't itself execute `poll()`, so the logic to respond to the safepoint is in both places.

```
1  volatile boolean flag = false;
2
3  void poll() {
4    if (flag) {
5      // wait for other threads
6      // execute action
7      // wait for other threads
8      // run deferred actions
9    }
10 }
11
12 void pauseAllThreadsAndExecute(Action action) {
13   flag = true;
14   // wait for other threads
15   flag = false;
16   // execute action
17   // wait for threads
18   // run deferred actions
19 }
```

Listing 7.8: Simple implementation of guest-language safepoints using a volatile flag.

A key limitation of the volatile flag technique is that the whole point of the volatile modifier is that it prevents the compiler from performing some optimisations such as caching the value instead of performing multiple reads. When our Ruby code is compiled and optimised by Graal we generally inline a very large

number of methods, as in Ruby all operators are method calls. This means that we are likely to end up with a large number of volatile reads in each method — at least one for each operator application — and the compiler will not attempt to consolidate them, as this is exactly what the volatile modifier is for.

7.5.2 Truffle Assumptions

In Chapter 6 we used Truffle's `Assumption` abstraction to cause a node to re-specialise to insert a debug action. To implement guest-language safepoints we can use the same primitive, but instead of re-specialising the AST based on an invalid assumption, we can just detect the invalidation, and then run the safepoint logic as already described in the implementation with the condition polling.

Listing 7.9 shows how the assumption is used. Instead of a volatile flag variable, we have a field holding the assumption. We annotate it as being `@CompilationFinal` so that when Graal compiles references to it, it will treat it as if it was a final field. The implementation of `poll` then just checks that the assumption is valid. If it is not, the same safepoint logic as described above is run.

```
1 // The assumption to implement the guest safepoint
2 @CompilationFinal Assumption assumption =
3     Truffle.getRuntime().createAssumption();
4
5 void poll() {
6     if (!assumption.isValid()) {
7         // safepoint logic
8     }
9 }
```

Listing 7.9: Implementation of guest-language safepoints with an `Assumption`.

On a conventional JVM, without the Graal compiler, the implementation of `Assumption` is pretty much as was shown for the volatile polling condition technique. However when the Graal compiler is available, assumption is implemented using dynamic deoptimisation, as was described in Chapter 6. When compiling a method that uses `poll()`, Graal will treat the assumption as always valid. The validity check will always succeed, and so to the Graal compiler the branch with the safepoint logic is dead code, and both are removed. This leaves us with no generated code from the `poll()` method.

To trigger the safepoint, the Truffle `Assumption` object is invalidated. Conceptually and on a conventional JVM we can say that this is equivalent to setting the flag variable as in the condition polling technique. When on Graal, invalidating the assumption will trigger deoptimisation. The compiled machine code for the `poll()` method does not actually do any checks, so we cause it to start doing so by transitioning to the interpreter.

Figure 7.1 shows how this implementation works in practice. Guest language application threads are represented as channels from left to right. At some point the topmost thread requests a safepoint in order to execute some method such as `each_object`. The safepoint assumption is invalidated, causing deoptimisation. There is a period of time while this happens for each thread, and for some threads it may take longer than others. The thread which takes longest to deoptimise is the one that defines the length of the latency for the safepoint. All threads are then within the guest-language safepoint and will execute the safepoint action concurrently. There is another barrier while all threads finish executing the action. Threads then begin executing normally again, and execute any deferred actions that were created while in the safepoint. Not shown is that threads continue executing in the interpreter and it may take time before they are optimised again and continue to execute at peak performance.

7.5.3 Switch Points

In our implementation we use a Truffle `Assumption` object, but there is actually an existing construct in the JVM that can achieve the same effect. The `SwitchPoint` class [81] in the `java.lang.invoke` package is part of the JVMs support for dynamic languages from JSR 292 [88], which also included the `invokedynamic` instruction.

Like `Assumption`, `SwitchPoint` includes a method to test if it is still valid, and a method to invalidate, and like `Assumption` when running on a JVM with the Graal compiler, it uses dynamic deoptimisation and existing GC safepoints to make testing validity zero-overhead.

JRuby [78], the implementation of Ruby on top of the JVM and a heavy user JSR 292, uses them for constant invalidation and for modifications of methods in existing classes.

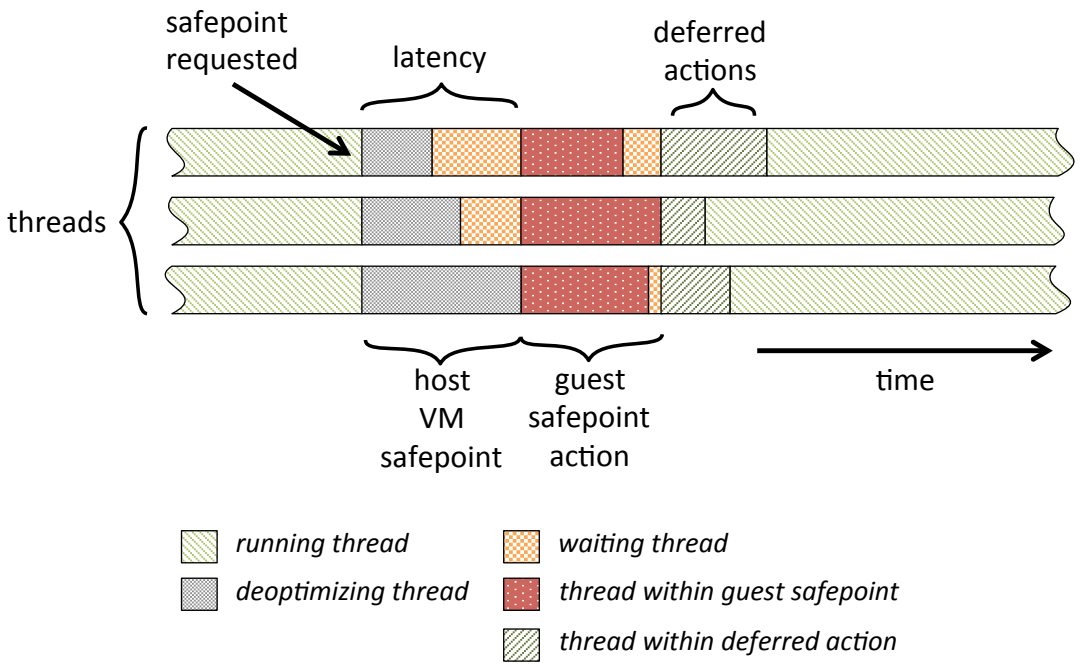


Figure 7.1: Phases of a guest-language safepoint

7.5.4 Blocking

There is an additional complication beyond the implementation that we have currently described. It will work with threads that are always running and always performing computation in the guest language, but they rely on the thread being able to actively respond to the request for a safepoint, and being able to run the safepoint logic and the action.

This may not always be the case, and there are at least three different conditions where a thread may be blocked so that it is not able to respond to a safepoint in the normal way.

First, threads may be blocked on trying to acquire a lock, waiting on a condition variable, a future, or some other concurrency primitive. Thankfully, the JVM provides us the `Thread.interrupt` method that allows a thread that is sleeping because it is blocked to be woken up. The blocking method that it was awaken in the middle of fails, with an exception thrown. Normally, calls that can throw this exception are run in a loop, with the operation retried, as the interrupts may be spurious. All we have to do to make this compatible with our guest-language safepoints implementation is to add a call to `Safepoints.poll()` at the head of this loop, but this should be done in all loops anyway. We then

add a call to `Thread.interrupt` on all guest language threads as part of the safepoint triggering logic.

Secondly, we have other blocking operations, but those which do not honour the `Thread.interrupt` system. In order to support as high a percentage of the Ruby language and core library as we do (see Subsection 3.3.5), we often need to make calls directly to the operating system rather than using the JVMs abstractions which are simpler but may not provide some key piece of functionality. If a thread is blocked on one of these low level calls, `Thread.interrupt` will not work. We do not currently have a perfect solution to this problem. One option is to run calls with timeouts of a few milliseconds and to `poll()` between calls in a loop. This will result in busy-waiting though, which may not scale well to a large number of blocked threads. Another option is to have threads dedicated to running blocking tasks in the background. Foreground application threads send a message to a background thread to perform a blocking call, and then wait on that background thread in an interruptible state. That way it is the background thread that is blocked, and the application thread remains able to respond to safepoints. However the implementation of this would be complicated and potentially introduce an overhead and reduce peak performance.

The third case is that of a thread that is running native code from a C extension. In conventional implementations of C extensions this would be the same problem as a low-level, and so uninterruptible, blocking call as just described. However, as will be described in Chapter 8, JRuby+Truffle supports C extensions through interpreting the source code of C extensions, rather than running the native code. In this case, our C interpreter can use the same guest-language safepoints API and include the same calls to `poll()`, and so will respond to requests for safepoints in exactly the same way as the Ruby code would.

7.6 Evaluation

7.6.1 Overhead on Peak Performance

We were initially interested in the overhead that JRuby+Truffle causes when our safepoint API is available but not used. This scenario is relevant because it is how we anticipate our safepoint mechanism to be used in most cases — always available but not a common operation. We also wanted to compare the overhead

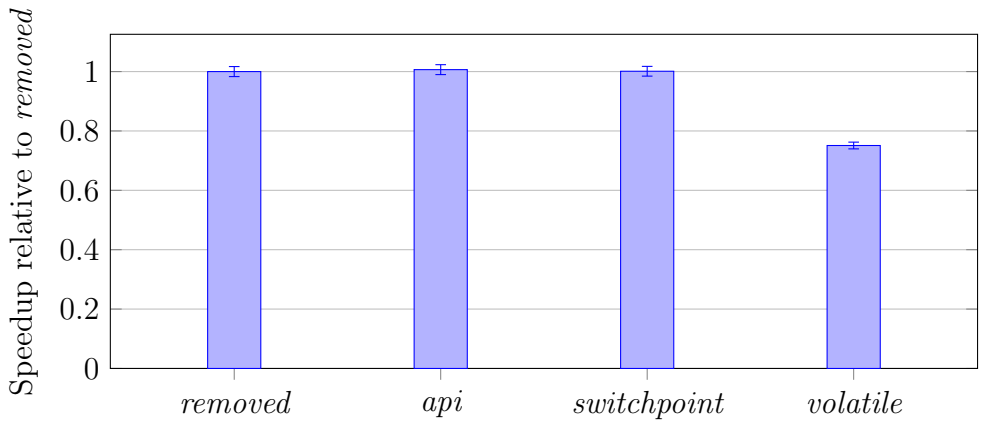


Figure 7.2: Peak performance of code with diferent safepoint implementations, normalized to the *removed* configuration.

of our safepoint API implementation with that of alternative implementations.

We first measured the performance of JRuby+Truffle including references to the safepoint API (the *api* configuration). Then we removed these references and measured again (the *removed* configuration). As the API is simple and compact, we only had to modify 74 lines of code to do this. We then measured the performance of JRuby+Truffle when using alternative safepoint implementation techniques. For example, we tried an implementation that explicitly checks a volatile flag (the *volatile* configuration). We also tried an implementation of our API that uses a JSR 292 SwitchPoint object (the *switchpoint* configuration) [88].

Figure 7.2 shows our results. There is no statistically significant difference in performance between *removed*, *api* and *switchpoint*. This is because both *api* and *switchpoint* are reusing the existing lower level VM safepoints, whose overhead is already part of the VM. Thus, our API adds no penalty to peak performance, and so we refer to it as *zero-overhead*.

There is, however, a statistically significant difference between the performance of *api* and *volatile*. The geometric mean of the overhead for the *volatile* configuration is $25\% \pm 1.5\%$. Of course, this is the overhead on the whole benchmarks and if we could compare the overhead of just the safepoint `poll()` operations it would be much greater.

7.6.2 Detailed Analysis

To further explain these results, we examined the machine code produced by the different configurations. We wrote a simple method that was run in an infinite loop so that it is dynamically optimised. Our example code, written in Ruby and executed by JRuby+Truffle, is shown in Listing 7.10. It is intentionally kept small to improve the readability of the machine code and just contains a few arithmetic instructions in the loop body to better show the effect of the different configurations. Every arithmetic operator in Ruby (in this case, `+`, `*` and `<`) is a method call. Therefore, any of these operations is a call site and conceptually does the check for guest-language safepoints. The body of the loop simply adds 7 to the counter i at each iteration until i becomes greater or equal to n. The method is called with different arguments to prevent *argument value profiling*, which would eliminate the whole loop entirely in compiled code as it has no side-effects.

```
1  def test(i, n)
2    while i < n
3      i += 1 + 2 * 3
4    end
5  end
6
7  while true
8    test(100, 200)
9    test(200, 300)
10 end
```

Listing 7.10: Example code for detailed analysis of the generated machine code.

We present the machine code with *symbolic names* for absolute addresses and rename the specific registers to the uppercase *name* of the variable they contain. We use the Intel syntax, in which the destination is the first operand.

The machine code produced by the *removed*, *api* and *switchpoint* configurations is identical if we abstract from absolute addresses and specific registers, and is shown in Figure 7.11. This supports our measurements and confirms that our API really has zero overhead, rather than just a low or difficult-to-measure overhead.

We can observe that the produced code is very close to the optimal code for such a loop. The operations in the body of the loop are reduced to a single

addition thanks to constant propagation. There are only two redundant *move* instructions, which copy the variable i between registers I and I'. The value of i is copied in I' to perform the addition because, if the `add` overflows, the original value of i needs to be accessed by the code performing the promotion to a larger integer type. In theory, the promotion code could subtract the second operand from the overflowed i, but this is a fairly complex optimisation to implement. The second *move* reunifies the registers.

The loop begins with a read on the safepoint polling page, which checks for VM safepoints[2]. In the *api* and *switchpoint* configurations, this check is also used for guest-language safepoints at no extra cost. After the `mov`, we `add` 7 to i and then check for overflow with `jo`, an instruction that jumps to the given address if there was an overflow in the last operation. We then have the second `mov`, followed by the loop condition $i < n$. The order of the operands in the machine code is reversed, so we must jump to the beginning of the loop if n is greater than or equal to i.

```
loop:
  test   safepoint polling page, eax # VM safepoint
  mov    I', I
  add    I', 0x7
  jo     overflow
  mov    I, I'
  cmp    N, I # n > i ?
  jg     loop
```

Listing 7.11: Generated machine code for the *api*, *removed* and *switchpoint* safepoint configurations.

We now look at the machine code produced by the *volatile* configuration (Figure 7.12). The generated code is much larger. The loop starts by testing the condition $i < n$, again with reversed operands. The condition is negated, $n \leq i$, as the test is to break out of the loop. Otherwise we enter the loop body. The body begins with 4 reads of the volatile flag from memory, and if it is found to be 0, the code jumps to a deoptimisation handler with `je`. Of these 4 checks, the first is for the loop itself and the other 3 are for the different calls to $+$, $+$ and $\times$ in the loop body. We then have the read on the safepoint polling page checking

[2]Actually, Graal moves this check out of the loop as it notices this is a bounded loop. We disabled that optimisation for clarity.

for VM safepoints. The remaining code is identical to Figure 7.11, except for the last two instructions. They perform a read on the volatile flag to check for guest-language safepoints at the call site of <, in the loop condition. If the flag is found to be valid, the control goes back to the beginning of the loop.

The 5 extra reads produced by the volatile flag are clearly redundant in the presence of the existing lower-level VM safepoints. They increase the number of instructions for the loop from 7 to 17, incurring a significant overhead as shown in Figure 7.2.

```
loop:
  cmp    N, I # n ≤ i ?
  jle    break out of loop
  cmp    VOLATILE FLAG, 0x0 # while loop safepoint
  je     deopt
  cmp    VOLATILE FLAG, 0x0 # i += 1
  je     deopt
  cmp    VOLATILE FLAG, 0x0 #      1 + 2
  je     deopt
  cmp    VOLATILE FLAG, 0x0 #          2 * 3
  je     deopt
  test   safepoint polling page, eax # VM safepoint
  mov    I', I
  add    I', 0x7
  jo     overflow
  mov    I, I'
  cmp    VOLATILE FLAG, 0x0 # i < n
  jne    loop
```

Listing 7.12: Generated machine code for the *volatile* safepoint configuration.

7.6.3 Overhead for Compilation Time

We also considered the time taken for dynamic compilation for benchmarks in different configurations by measuring the time taken to compile the main method from the Mandelbrot benchmark. This is a relatively large method with a high number of method calls which need to be inlined and several nested loops, all of which add guest safepoint checks.

Figure 7.3 shows our results, with the columns showing mean compilation and the error bars showing one standard deviation. We found no significant difference

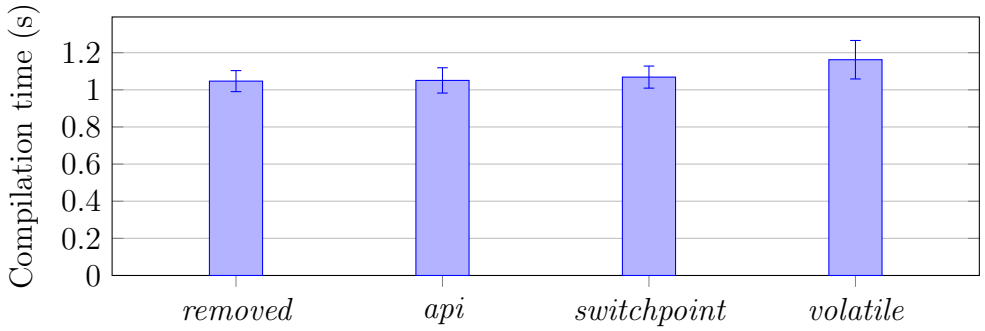


Figure 7.3: Mean compilation time for the Mandelbrot method across different configurations.

in compilation time between *removed*, *api* and *switchpoint*. Compilation time for *volatile flag* appeared to be only slightly higher. All of the techniques explored require extra work in the compiler due to extra `poll()` calls, but this appears to be insignificant compared to the rest of the work being done by the compiler. The *volatile flag* is different to the other implementations in that the code is not removed in early phases and adds extra work to later phases of the compiler.

7.6.4 Latency

Finally, we considered the time it takes for all threads to reach a guest-language safepoint after one thread requested it — the *latency*. Comparing the different configurations is not relevant here, as the costs can be primarily attributed to the VM safepoint latency and the necessary deoptimisations to run the omitted Java code, replaced in the compiled code by a VM safepoint check.

We ran the Mandelbrot benchmark with a variable number of threads. After steady state was reached, a separate thread requested all others to enter a guest safepoint.

Figure 7.4 shows our results, with the columns showing mean latency and the error bars showing one standard deviation. Latency was reasonable for most applications, responding to requests for a guest safepoint within about 1/100th of a second when running 8 threads. Variance was surprisingly high, which would make it hard to provide performance guarantees. Latency increases with the number of threads running concurrently, however the trend is sublinear (for 64 threads the latency is just 3× that for one thread). Deoptimisation for multiple threads is a parallel task, so although multiple threads add extra work, they also

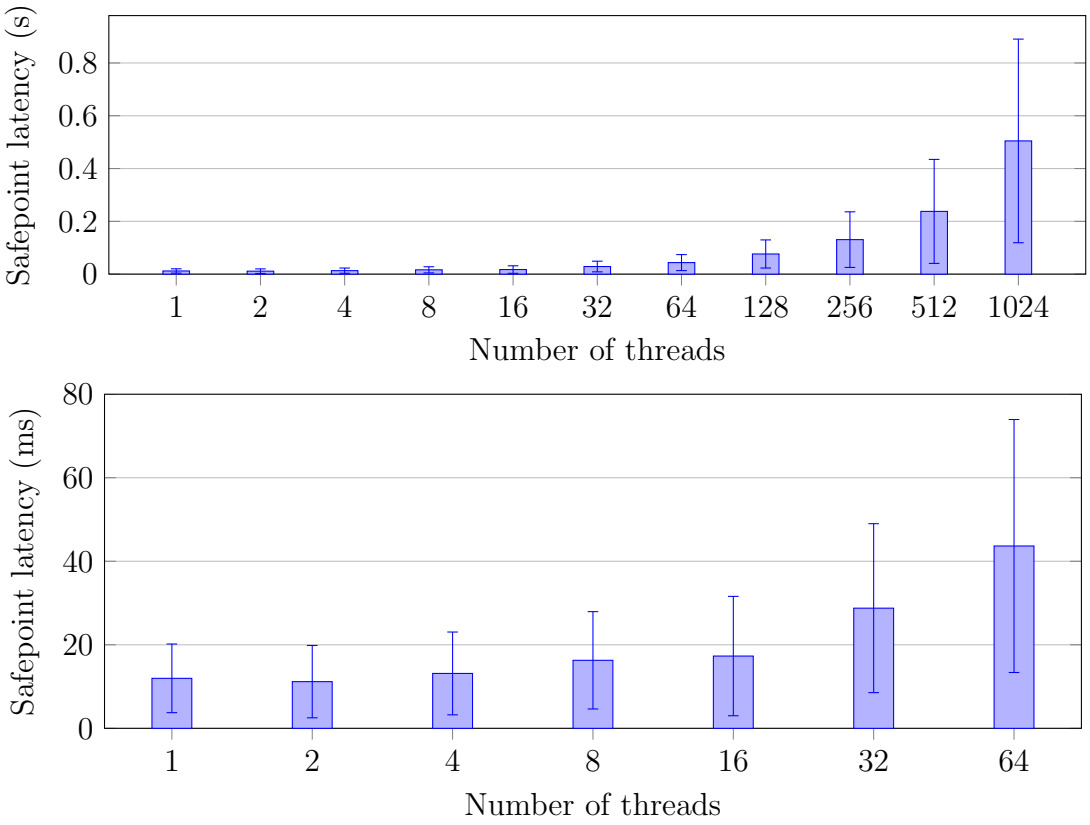


Figure 7.4: Safepoint latency for the Mandelbrot for our implementation.

add extra potential parallelism. For 1024 threads, a number well beyond typical for normal modern Ruby applications, latency was half a second. It should also be noted that due to deoptimisation, the use of a guest safepoint brings peak performance down for some time before code can be reoptimised. Very frequent use of guest safepoints could cause an application to run entirely in the interpreter.

Deoptimisation of the call stacks of different threads is implemented in the underlying JVM to run in parallel using multiple cores if they are available. These experiments were run on a system with two processors with four cores each, so it should be possible to deoptimise up to eight threads in parallel. As the results have a fairly high error margin, it does not appear to be possible to make any connection between the parallelism available in the system and the safepoint latency.

In the current implementation, using a safepoint triggers global deoptimisation. In a large application which has taken time to warm up, this could be a lot of code being invalidated. However, Truffle and Graal do attempt to reduce the cost in a couple of ways. Truffle will retain profiling information about the program, so after deoptimisation profiled types, branch probabilities and so on are still available. Method invocation and loop counts are also retained so that the threshold to re-optimise is very low. Finally, Graal can cache IR from different stages of compilation [99]. In the future this could be used to more quickly compile methods where the ASTs have not changed since the last compilation.

7.7 Summary

We have shown how to use the existing safepoint system in a JVM and make it available, with greater flexibility, to implementations of guest languages. We have shown multiple diverse applications of this technique in the context of the Ruby programming language.

Crucially, we have demonstrated that it has zero-overhead on peak temporal performance when the safepoints are available but not used, and when the program is stable and has been optimised as described in Chapter 4. The underlying VM implementation of safepoints is already needed to support services such as GC, and our re-use of them does not change how frequently potential safepoints need to be inserted into the instruction stream. This means that we can use our guest-language safepoints to provide the demonstrated Ruby language features,

always enabled.

Although our implementation is based on functionality provided by Truffle and the Graal backend for Truffle, we also showed how the technique can also be implemented with functionality already available in a standard JVM.

Although it is out of the scope we set ourselves in Chapter 4, in this chapter we did examine some additional metrics, such as the overhead on compilation time and the latency for responding to a request for a safepoint and found them both to be acceptable.

Chapter 8

Interpretation of Native Extensions for Dynamic Languages

8.1 Introduction

Due to the low performance of most existing implementations of Ruby, even lower performance in the previous generation of implementations, and low relative performance of language features which use metaprogramming, the Ruby community has tended to write C extensions for applications where performance is important.

A C extension is a dynamic library, normally written using C but really it could be written using any language which can export functions using the same ABI, such as C++. A Ruby implementation such as MRI or Rubinius loads and dynamically links the compiled library into the implementation. An initial call is made into the library, which can then register Ruby modules, classes, methods and so on, as if they had been defined in Ruby but backed by a C function.

The C code needs some way to interact with the Ruby interpreter, so MRI provides an API with a header file to access Ruby values, call methods and other operations. Listing 8.1 shows an example function from this API, the `rb_ary_store` function, allows a C extension to store an element into a Ruby array.

This model for C extensions works well for the original implementation of Ruby. As the API directly accesses the implementation's internal data structures, the interface is powerful, has low overhead, and was simple for MRI to add: all

```
typedef void* VALUE;
void rb_ary_store(VALUE ary, long idx, VALUE val);
```

Listing 8.1: Function to store a value into an array as part of the Ruby API.

they had to do was make their header files public and support dynamic loading of native modules.

The problem is that the extension API exposes almost the entire internal structure of MRI. The header files that C extensions use are literally the internal MRI header files. Even if we only consider MRI, this means that few changes can be made to the internal structure of the implementation without breaking this interface. This limits the innovation possible in MRI. The changes that were introduced with the YARV bytecode interpreter for MRI 1.9 have been the most significant attempted, and they only changed the execution model, not the major internal data structures.

The problem is much worse for alternative implementations of Ruby that want to provide optimisations. They have to find a way to provide exactly the same API and somehow abstract over their optimisations.

As a concrete example, JRuby+Truffle provides optimised multiple implementations of the Ruby `Array` class, depending on the values stored in it. We apply the storage strategies pattern [17] to represent an array that only contains small integer values with a `int[]`. If some of the values are larger than 64 bits we may use a `long[]` instead. We also support `double[]` for `Float` values, and a generic `Object[]` for anything else or for heterogeneous arrays. This is a significant optimisation as it reduces boxing allocations, and is key for our performance.

However, the Ruby C API has a macro to get the raw storage pointer that backs a Ruby array, `VALUE* RARRAY_PTR(array)`. This returns a `VALUE*`, which is an array of Ruby's internal tagged union for representing any object. When a C library has this pointer it can access values directly. The problem that will be solved in this chapter is, how we provide the API that the C extensions expect, which use an array of tagged unions, when we want to optimise the implementation to use multiple storage strategies.

In the context of MRI, C extensions provide a very good solution to the problem of limited performance, as the implementation of MRI is so limited. However with more sophisticated implementations of Ruby that apply optimisations, native calls may actually be a barrier to performance. Implementations such as

JRuby+Truffle and Topaz achieve high performance through aggressive and deep inlining and being able to trace objects through complicated control flow to see where they are used and remove allocations and propagate constant values. A C extension with a native call prevents these optimisations, as the dynamic compiler can only see the C extension method as an opaque function pointer. This problem is also addressed by the techniques in the chapter.

C extensions are prevalent in the Ruby ecosystem, and anecdotally most large Ruby applications will probably depend on at least one C extension at some point in their deep chain of dependencies. We surveyed 421,817 Ruby modules (known as gems) available in the RubyGems repository. Among these, almost 14% have a transitive dependency on a gem containing a C extension—although without actually running the gems it is not possible to clarify whether the C extensions are optional or required.

An alternate reason that people write C extensions for Ruby is to access existing native libraries, such as database drivers. This is not the key use case considered here, and we do not evaluate for this use case, but we do mention how we can support it at the end of this chapter.

The research contributions in this chapter were collaborative work with Matthias Grimmer at Johannes Kepler Universität, Linz. The interpreter for C was existing work [44], and work to integrate with the Ruby interpreter to support C extensions was joint work, but research contributions including support for inner pointers from C to Ruby objects, representing high-level Ruby arrays as C pointers, support for library routines such as `malloc` on Ruby objects and cross-language optimisations were all novel research contributions by the author of this thesis. The collaborative work described in this chapter will also form part of the doctoral thesis of Matthias Grimmer.

8.2 Existing Solutions

This problem has been acknowledged for a long time, in multiple languages. Python provides exactly the same kind of C extension API that exposes the internals of the reference implementation, and the key alternative implementation in that community, PyPy, has struggled to gain acceptance as it does not provide any support for C extensions due to this problem [6]. In Python the problem is actually worse, as the Python API dictates a reference counting GC algorithm,

which may not be easily compatible with more advanced tracing collectors used by optimised implementations.

Some solutions to this problem are not realistic due to the scale at which these languages are used and the number of existing C extensions which people want to run. For example, both Ruby and Python provide a Foreign Function Interface (FFI) with routines for dynamically linking and calling compiled C code and for marshalling values in C's type system. Here there is no need for a wrapper using the language specific API as C functions are called directly and guest-language values and data structures are marshalled by the application. However these FFIs have come later in the language development, and asking developers to rewrite their C extensions so that the FFI can use them is not realistic due to the large body of existing code.

One more realistic solution is to implement a bridging layer between the API that C extensions expect and the actual internal implementation of the language. However, this introduces costs for lowering the optimised data structures used by the more modern implementation in order to allow them to be accessed using the existing API. Performance is usually one goal of using C extensions, so adding a bridging layer is not ideal.

Rubinius supports C extensions through such a compatibility layer. This means that in addition to problems that MRI has with meeting a fixed API, Rubinius must also add another layer that converts routines from the MRI API to calls on Rubinius' C++ implementation objects. The mechanism Rubinius uses to optimise Ruby code, an LLVM-based JIT compiler, cannot optimise through the initial native call to the conversion layer. At this point many other useful optimisations no longer can be applied. Despite having a significantly more advanced implementation than MRI, Rubinius actually runs C extensions about half as fast as MRI (see Section 8.10). This is clearly at odds with the whole point of C extensions in most instances.

JRuby used to provide limited experimental support for C extensions until this was removed after the work proved to be too complicated to maintain and the performance too limited [2, 5]. Their implemented used the JVMs FFI mechanism, JNI, to call C extensions. This technique is almost the same as used in Rubinius, also using a conversion layer, except that now the interface between the VM and the conversion layer is even more complex. For example, the Ruby C API makes it possible to take the pointer to the character array representing

a string. MRI and Rubinius are able to directly return the actual pointer, but in JRuby using JNI it is not possible to obtain the address of the character array in a string. In order to implement this routine, JRuby must copy the string data from the JVM onto the native heap. When the native string data is then modified, JRuby must copy it back into the JVM. To keep both sides of the divide synchronized, JRuby must keep performing this copy each time the interface is passed. We believe that this is the cause for the benchmarks in our evaluation timing out for JRuby and Rubinius.

These implementations of C extensions are not just limited, but in the case of JRuby where the requirement to copy can balloon, they are in some case intractable.

8.3 TruffleC

Our solution is to combine the Truffle implementation of Ruby, JRuby+Truffle, with an existing Truffle implementation of C, TruffleC [44, 42]. Truffle dynamically executes C code on top of a JVM and performs well compared to industry standard C compilers such as GCC or LLVM/Clang in terms of peak-performance. While it may seem unusual to talk of an interpreter for C, C is a relatively simple language and apart from unrestricted native access to the heap, most of the language features such as arithmetic and control flow are not much different than are found in a language which is typically interpreted.

Despite C being a static language, TruffleC uses the *self-optimisation* capability of Truffle: It uses PICs to efficiently handle function pointer calls, profiles branch probabilities to optimistically remove never executed code, or profiles runtime values and replaces them by constants if they do not change over time. TruffleC also has the ability to access native C libraries of which no source code is available, using the *Graal native function interface* [45]. This interface can directly access native functions from Java. However this functionality is not used in this evaluation.

8.4 Language Interoperability on Top of Truffle

Existing work describes the novel idea of a cross-language mechanism that allows us to compose arbitrary interpreters efficiently on top of Truffle [43].

The goal for this mechanism was to retain the modular way of implementing languages on top of Truffle and to meet the following criteria:

- Languages can be treated as modules and are composable. An implementation of a cross-language interface, such as C extensions for Ruby, requires very little effort because any two languages that support this mechanism are already interoperable.

- We do *not* want to introduce a new object model that all Truffle guest languages have to share, which is based on memory layouts and calling conventions. Although some languages, such as Python and Ruby, have superficially similar object models, a shared object model is not applicable to a wider set of languages. For example, JRuby+Truffle uses a specific high-performance object model [112] to represent Ruby runtime data, whereas TruffleC stores C runtime data such as arrays and structures directly on the native heap as is suitable for the semantics of C. We introduce a common *interface* for objects that is based on code generation via ASTs. Our approach allows sharing language specific objects (with different memory representations and calling conventions) across languages, rather than lowering all objects to a common representation.

- We want to make the language boundaries completely transparent to Truffle's dynamic compiler, in that a cross-language call should have exactly the same representation as an intra-language call. This transparency allows the JIT compiler to inline and apply advanced optimisations such as constant propagation and escape analysis across language boundaries without modifications.

In the following sections we describe in detail how we extend the Truffle framework with this mechanism. We use the mechanism to access Ruby objects from C and to forward Ruby API calls from the TruffleC interpreter back to the JRuby+Truffle interpreter. Therefore our system includes calls both from Ruby to C and from C back to Ruby.

Using ASTs as an internal representation of a user program already abstracts away syntactic differences of object accesses and function calls in different languages. However, each language uses its own representation of runtime data such as objects, and therefore the access operations differ. Our research therefore

focused on how we can share such objects with different representations across different interpreters.

In this system we call every non-primitive entity of a program an *object*. This includes Ruby objects, classes, modules and methods, and C immediate values and pointers. An object that is being accessed by a different language than the language of its origin is called a *foreign object*. A Ruby object used by a C extension is therefore considered foreign in that context. If an object is accessed in the language of its origin, we call it a *regular object*. A Ruby object, used by a Ruby program is therefore considered regular. Object accesses are operations that can be performed on objects, e.g. method calls or property accesses.

8.4.1 Language-independent Object Accesses

In order to make objects (objects that implement `TruffleObject`) *shareable* across languages, we require them to support a common interface. We implement this as a set of *messages*:

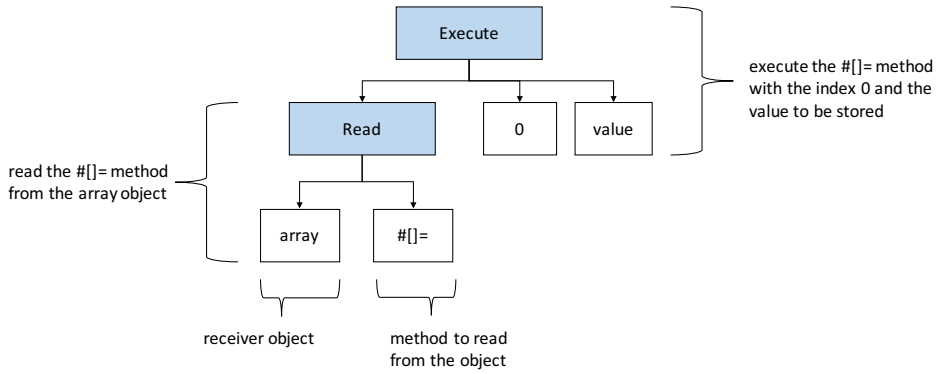
Read: We use the *Read* message to read a member of an object denoted by the member's identity. For example, we use the *Read* message to get properties of an object such as a field or a method, and to read elements of an array.

Write: We use the *Write* message to write a member of an object denoted by its identity. Analogous to the *Read* message, we use it to write object properties.

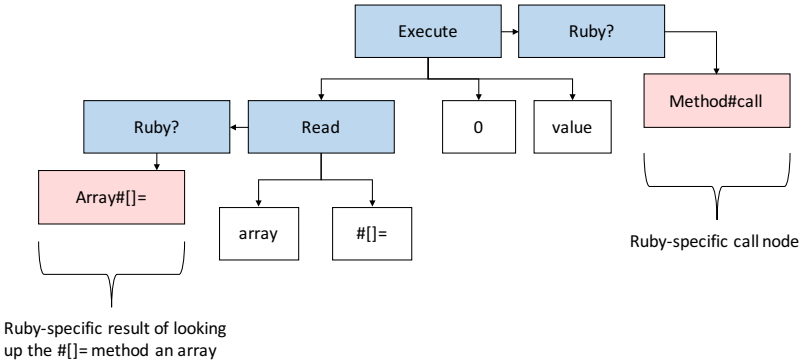
Execute: The *Execute* message, which can have arguments, is used to evaluate an object. For example, it can evaluate a Ruby method or invoke the target of a C function pointer.

Unbox: If the object represents a boxed numeric value and receives an *Unbox* message, this message unwraps the boxed value and returns it. For example, if an *Unbox* message is sent to a Ruby Fixnum, the object returns its value as a 4 byte integer value.

We call an object *shareable* if we can access it via these language-independent messages. Truffle guest-language implementations can insert language-independent message nodes into the AST of a program and send these messages in order to access a foreign object. Figure 8.1a shows an AST that accesses a Ruby array



(a) Using messages to read the Ruby `[]=` method, and then call it with an index and value to be stored



(b) The same AST with the messages resolved to Ruby-specific operations and a language guard

Figure 8.1: Language independent object access via messages.

via messages in order to store `value` at index 0. This interpreter first sends a *Read* message to get the array setter function `[]=` from the array object (in Ruby writing to an element in an array is performed via a method call). The language-independent nodes are shown in blue. Afterwards it sends an *Execute* message to evaluate this setter function. Figure 8.1a shows how these messages are resolved to Ruby-specific nodes in red the first time they are run. Each language-independent node is resolved to the node to perform this action in the language of the given object. As another object from a different language may be seen next time, a guard is added between the language-independent and language-specific nodes that checks the object is from the expected language. This is similar to how a PIC checks the type of a receiver object before resolving to the correct method. If this check failed, then another language could be added onto the end

of this chain. Again, this is similar to how a PIC handles multiple receiver types at a single call site. This approach gives us a technique which could be called *language polymorphic* call sites.

Given this mechanism, Truffle guest languages can access any foreign object that implements this message-based interface. If an object does not support a certain message we report a runtime error with a high-level diagnostic message.

8.4.2 Message Resolution

The receiver of a cross-language message does *not* return a value that can be further processed. Instead, the receiver returns an AST snippet — a small tree of nodes designed for insertion into a larger tree. This AST snippet contains language-specific nodes for executing the message on the receiver. *Message resolution* replaces the AST node that sent a language-independent message with a language-specific AST snippet that directly accesses the receiver. After message resolution an object is accessed directly by a receiver-specific AST snippet rather than by a message.

During the execution of a program the receiver of an access can change if it is a non-final value, and so the target language of an object access can change as well. Therefore we need to check the receiver's language before we directly access it. If the foreign receiver object originates from a different language than the one seen so far we access it again via messages and do the message resolution again. If an object access site has varying receivers, originating from different languages, we call the access *language polymorphic*. To avoid a loss in performance, caused by a language polymorphic object access, we embed AST snippets for different receiver languages in a chain similar to a conventional inline cache [54], except that here the cache handles multiple languages as well as multiple classes of receivers.

Message resolution and building object accesses at runtime has the following benefits:

Language independence: Messages can be sent to any shareable object. The receiver's language of origin does not matter and messages resolve themselves to language-specific operations at runtime. This mechanism is not limited to C extensions for Ruby but could possibly be used for many combination of languages.

No performance overhead: Message resolution only affects the application's

performance upon the first execution of an object access for a given language. Once a message is resolved and as long as the languages used remain stable, the application runs at full speed.

Cross-language inlining: Message resolution allows the dynamic compiler to inline methods even across language boundaries. By generating AST snippets for accessing foreign objects we avoid the barriers from one language to another that would normally prevent inlining. Our approach creates a single AST that merges different language-specific AST parts. The language-specific parts are completely transparent to the JIT compiler. Removing the language boundaries allows the compiler to inline method calls even if the receiver is a foreign object. Widening the compilation unit across different languages is important [97, 13] as it enables further optimisations such as specialization and constant propagation.

8.4.3 Shared Objects and Shared Primitive Values

Like a regular object access, a foreign object access produces and returns a result. Our interoperability mechanism distinguishes between two types of values that a foreign object access can return:

Object types: If a foreign object access returns a non-primitive value, this object again has to be *shareable* in the sense that it understands the messages *Read*, *Write*, *Execute*, and *Unbox*. If the returned object is accessed later, it is accessed via these messages.

Primitive types: In order to exchange primitive values across different languages we define a set of *shared primitive types*. We refer to values with such a primitive type as *shared primitives*. The primitive types include signed and unsigned integer types (8, 16, 32 and 64 bit versions) as well as floating point types (32 and 64 bit versions) that follow the IEEE floating point 754 standard. The vast majority of languages use some of these types, and as they are provided by the physical architecture their semantics are usually identical. In the course of a foreign object access, a foreign language maps its language-specific primitive values to *shared primitive* values and returns them as language-independent values. When the host language receives a *shared primitive* value it again provides a mapping to host language-specific values.

8.4.4 JRuby+Truffle: Foreign Object Accesses and Shareable Ruby Objects

In Ruby's semantics there are no non-reference primitive types and every value is logically represented as an object, as in the tradition of languages such as Smalltalk. Also, in contrast to other languages such as Java, Ruby array elements, hash elements, or object attributes cannot be accessed directly but only via getter and setter calls on the receiver object. For example, a write access to a Ruby array element is performed by calling the `[]=` method of the array and providing the index and the value as arguments.

In our Ruby implementation all runtime data objects as well as all Ruby methods are shareable in the sense that they implement our message-based interface. Figure 8.1 shows how a Ruby array can be accessed via messages.

Ruby objects that represent numbers, such as `Fixnum` and `Float` that can be simply represented as primitives common to many languages, also support the *Unbox* message. This message simply maps the boxed value to the relative shared primitive. For example, a host language other than Ruby might send an *Unbox* message whenever it needs the object's value for an arithmetic operation.

8.4.5 TruffleC: Foreign Object Accesses and shareable C Pointers

TruffleC can share primitive C values, mapped to *shared primitive values*, as well as pointers to C runtime data with other languages. In our implementation, pointers are objects that implement the message interface, which allows them to be shared across all Truffle guest language implementations. TruffleC represents all pointers (so including pointers to values, arrays, structs or functions) as `CAddress` Java objects that wrap a 64-bit value [48]. This value represents the actual referenced address on the native heap. Besides the address value, a `CAddress` object also stores type information about the referenced object. Depending on the type of the referenced object, `CAddress` objects can resolve the following messages:

- A pointer to a C struct can resolve *Read*/*Write* messages, which access members of the referenced struct.
- A pointer to an array can resolve *Read*/*Write* messages that access a certain

array element.

- Finally, `CAddress` objects that reference a C function can be executed using the *Execute* message.

When JRuby+Truffle accesses a function that is implemented within a C extension, it will use an *Execute* message to invoke it. Message resolution will bridge the gap between both languages automatically at runtime. The language boundaries are transparent to the dynamic compiler and it can inline these C extension functions just like normal Ruby functions.

TruffleC allows binding foreign objects to pointer variables declared in C. Hence, pointer variables can be bound to `CAdress` objects as well as shared foreign objects. TruffleC can then dereference these pointer variables via messages.

8.5 Implementation of the Ruby C API

Developers of a C extension for Ruby access the API by including the `ruby.h` header file. We want to provide the same API as Ruby does for C extensions, i.e., we want to provide all functions that are available when including `ruby.h`. To do so we created our own source-compatible implementation of `ruby.h`. This file contains the function signatures of all of the Ruby API functions that were required for the modules we evaluated, as described in the next section. We believe it would be tractable to continue the implementation of API routines so that the set available is reasonably complete.

Listing 8.2 shows an excerpt of this header file.

We do not provide an implementation for these functions in C code. Instead, we implement the API by substituting every invocation of one of the functions at runtime with a language-independent message send or directly access the Ruby runtime.

We can distinguish between *local* and *global* functions in the Ruby API:

8.5.1 Local Functions

The Ruby API also offers a wide variety of functions that are used to access and manipulate Ruby objects from within C. In the following we explain how we substitute the Ruby API functions `rb_ary_store`, `rb_iv_get`, `rb_funcall`, and `FIX2INT`:

```
typedef VALUE void*;
typedef ID void*;

// Define a C function as a Ruby method
void rb_define_method
(VALUE class, const char* name,
VALUE(*func)(), int argc);

// Store an array element into a Ruby array
void rb_ary_store
     (VALUE ary, long idx, VALUE val);

// Get the Ruby internal representation of an
// identifier
ID rb_intern(const char* name);

// Get instance variables of a Ruby object
VALUE rb_iv_get(VALUE object,
    const char* iv_name)

// Invoke a Ruby method from C
VALUE rb_funcall(VALUE receiver ID method_id,
    int argc, ...);

// Convert a Ruby Fixnum to C long
long FIX2INT(VALUE value);
```

Listing 8.2: Excerpt of the `ruby.h` implementation.

- `rb_ary_store`:
  Normally TruffleC would insert *call nodes* for regular function calls, however, TruffleC handles invocations of these API functions differently. Consider the invocation of the `rb_ary_store` function (Listing 8.3): Instead of a call node, TruffleC inserts message nodes that are sent to the Ruby array (`array`). The AST of the C program (Listing 8.3) now contains two message nodes (namely a *Read* message to get the array setter method `[]=` and an *Execute* message to eventually execute the setter method, see Figure 8.1a). Upon first execution these messages are resolved (Figure **??**), which results in a TruffleC AST that embeds a Ruby array access (Figure 8.1b).

- `rb_iv_get`:

```
1 VALUE array = ... ; // Ruby array of Fixnums
2 VALUE value = ... ; // Ruby Fixnum
3
4 rb_ary_store(array, 0, value);
```

Listing 8.3: Calling `rb_ary_store` from C.

In contrast to Ruby object attributes, which are accessed via getter and setter methods, Ruby instance variables can be accessed directly. Therefore the substitution of `rb_iv_get` sends a *Read* message and provides the name of the accessed instance variable.

- `rb_funcall`:
  The function `rb_funcall` allows calling a Ruby method from within a C function. We substitute this call again by two messages. The first one is a *Read* message that resolves the method from the Ruby receiver object. The second message is an *Execute* message that finally executes the method.

- `FIX2INT`:
  We replace functions that convert numbers from Ruby objects to C primitives (such as `FIX2INT`, Fixnum to integer) by *Unbox* messages, sent to the Ruby object (`VALUE value`). As the naming convention suggests, `FIX2INT` is usually implemented as a C preprocessor macro. For the gems we studied this difference did not matter, and if it did we could implement it as a macro that simply called another function.

8.5.2 Global Functions

The Ruby API offers various different functions that allow developers to manipulate the global object class of a Ruby application from C or to access the Ruby engine.

The API includes functions to define global variables, modules, or global functions (e.g., `rb_define_method`) etc. Also, these functions allow developers to retrieve the Ruby internal representation of an identifier (e.g. `rb_intern`). In order to substitute invocations of these API functions, TruffleC accesses the global object of the Ruby application using messages or directly accesses the Ruby engine.

For instance, we substitute calls to `rb_define_method` and `rb_intern` as follows:

- `rb_define_method`:
  To define a new method in a Ruby class, developers use the `rb_define_method` function. TruffleC substitutes this function invocation and sends a *Write* message to the Ruby class object (first argument, `VALUE class`). The Ruby class object resolves this message and adds the C function pointer (`VALUE(*func)()`) as a new method. The function pointer (`VALUE(*func)()`) is represented as a `CAddress` object. When invoking the added function from within Ruby, JRuby+Truffle uses an *Execute* message and can therefore directly invoke this C function.

- `rb_intern`:
  `rb_intern` converts a C string to a reference to a shared immutable Ruby object which can be compared by reference to increase performance. TruffleC substitutes the invocation of this method and directly accesses the JRuby+Truffle engine. JRuby+Truffle exposes a utility function that allows resolving these immutable Ruby objects.

Given this implementation of the API we can run C extensions without modification and are therefore compatible with the Ruby MRI API. Given the interoperability mechanism presented in this paper, implementing this API via message-based substitutions was a trivial task: The implementation of TruffleC simply uses the interoperability mechanism and replaces invocations of Ruby API methods with messages. Besides these changes, no modifications of JRuby+Truffle or TruffleC were necessary to support C extensions for Ruby. This demonstrates that our cross-language mechanism is applicable in practice and makes language compositions easy.

8.6 Expressing Pointers to Managed Objects

The Ruby C API provides several functions which give direct access to the internals of data structures via pointers. This is often done for performance reasons. For example we already described how it is possible to get the `VALUE*` that implements a Ruby array. It's then possible to iterate over this pointer without using any further Ruby API calls, which may be faster than using an API call to access each element of an array. Similar pointer access is commonly used for the Ruby `String` class, which provides access to the underlying `char*`.

Implementing this pointer access is extremely problematic for existing implementations of Ruby that apply optimisations as they will likely want to use more sophisticated representation for data structures, but they have to meet the same API as MRI uses. We previously gave array storage strategy specialisation as an example of an optimisation that pointer access conflicts with.

In the C extension support in JRuby+Truffle, a normal pointer to a Ruby object is modelled in C as a simple Java reference to a *sharable* `TruffleObject`. If additional indirection is introduced and a pointer is taken to a Ruby object, TruffleC creates a `TruffleObjectReference` object (which itself is sharable) that wraps the object that is pointed to. Additional wrappers can achieve arbitrary levels of pointer indirection. As with any object that does not escape, as long as the pointer objects are created and used within a single compilation unit (which may of course include multiple Ruby and C methods) the compiler can optimise and remove these indirections (see Section 8.10) and thus do not introduce a time or space overhead.

When the pointer is dereferenced, a method is called on the sharable object to return the byte at a given offset within the object. In this way, the object can be dynamically asked to return the memory that would be at an address, if the object were implemented in MRI. The JRuby+Truffle `String` class can then return individual characters that would be at an offset if it were implemented as a simple `char*`, even though in reality it is implemented as a managed Java array.

C also allows arithmetic on pointers, and arithmetic on internal pointers to Ruby objects is exercised in some of our benchmarks. To support this in TruffleC a second wrapper, a sharable `TruffleObjectReferenceOffset` object, holds both the object reference (`TruffleObjectReference`) and the offset from that address. Further pointer arithmetic just produces a new wrapper with a new offset. Any dereference of this object-offset wrapper will use the same messages to read or write from the object as a normal array access would.

8.7 Memory Operations on Managed Objects

The C extensions that we experimented with not only used pointer arithmetic, but then passed those pointers to standard library routines such as `memcpy`, a routine which copies a number of bytes from one pointer to another.

To implemented this we added specialisations for common memory routines which use the same mechanism as described in the previous section to ask the object to return the byte that would be at an offset if the object was not managed. A `memcpy` into a managed object, rather than out of it, works in a similar way. Memory operations outside the bounds of the object are not supported and throw an exception.

8.8 Limitations

TruffleC only supports well-defined programs. Many parts of the C language that would be more difficult to support are explicitly left undefined by the standard [108], so a valid implementation would be to throw an exception, or any other action. Some other parts of the language are explicitly optional, so TruffleC could still be a conforming implementation while not supporting them. Examples of these operations include casts between different types of pointer, or between pointers and integers. These are undefined at the point of dereferencing. TruffleC does not support this and will report a runtime error. As these operations are left undefined or optional by the C standard, this is a valid implementation.

In practice, this technical conformance may not be sufficient for other languages where we would also want to support C extensions, as they may rely on the actual behaviour of operations in common compilers, even if they are technically undefined.

8.9 Cross-Language Optimisations

So far we have discussed techniques that allow JRuby+Truffle to run C extensions without conflicting with optimisations that we are using to achieve high performance and without the massive overhead of marshalling data back and forth from managed to native heaps that limits the JRuby and Rubinius implementations. However, an implementation of C extensions where C and Ruby are both implemented using a common dynamically optimising framework introduces new optimisation opportunities.

The Truffle's Graal backend only knows about ASTs, not different languages, so a method call that crosses a language barrier can be inlined in exactly the same way as a call that is from one language to the same language. This is

heavily exercised in the benchmarks which we evaluated below, where Ruby code may call a C extension, which calls back into the Ruby code. Truffle is able to inline through all three levels of call and through both languages, producing a single piece of machine code for the original Ruby method and everything it calls. Inlining is a key optimisation, and this then enables further optimisations such as cross-language escape analysis and allocation removal. Objects which are allocated in Ruby and passed into the C extension can be optimised away in the same way as if both methods were written in Ruby.

8.10 Evaluation

8.10.1 Benchmarks

In this chapter we focus on the benchmarks from Chunky PNG [107] and PSD.rb [69], as they are available in both the form of pure Ruby and the C extensions with the Oily PNG [67] and PSD Native [68] libraries.

8.10.2 Required Modifications

Running these gems on our system required just one minor modification for compatibility: we had to replace two instances of stack allocation of a variable-sized array with heap allocation via `malloc` and `free`. Variable-sized array allocation on the stack is a feature from C99 which TruffleC does not support yet.

We also had to fix two bugs where an incorrect type was used, `int` instead of `VALUE`, causing a tagged integer to be used as if it was untagged. This was an implementation bug in the gem rather than a TruffleC incompatibility, as it was causing a different result between the Ruby module and the C extension on all Ruby language implementations. However, TruffleC was able to differentiate between the two types where the error was missed by the original authors when running with GCC[1].

Apart from these minor modifications our system runs all the methods from these two significant C extensions, unmodified.

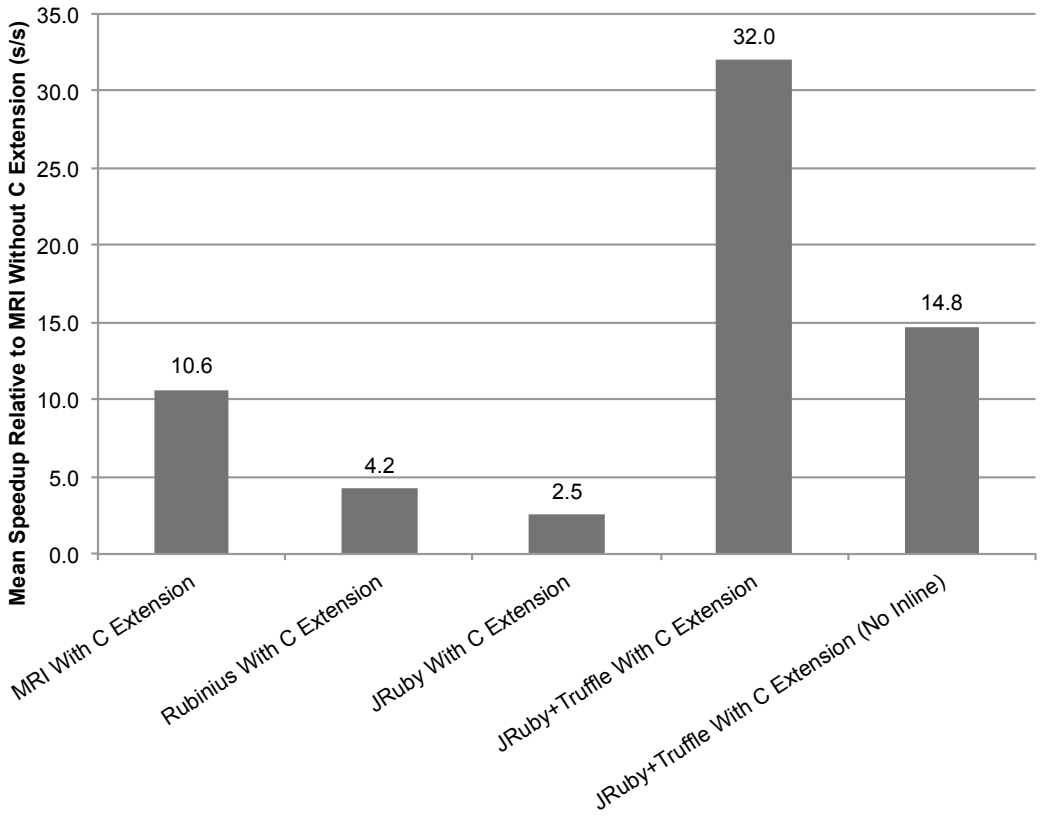


Figure 8.2: Summary of speedup across all native extensions.

8.10.3 Results

Figure 8.2 shows a summary of our results. We show the geometric mean speedup of each evaluated implementation over all benchmarks, relative to the speed at which MRI ran the Ruby code without the C extension. When using MRI the average speedup of using the C extension (*MRI With C Extension*, Figure 8.2) over pure Ruby code is around 11×. Rubinius (*Rubinius With C Extensions*, Figure 8.2) only achieves around one third of this speedup. Although Rubinius generally achieves at least the same performance as MRI for Ruby code, its performance for C extensions is limited by having to meet MRI's API, which requires a bridging layer. Rubinius failed to make any progress with 3 of the benchmarks in a reasonable time frame so they were considered to have timed out. The performance of JRuby (*JRuby With C Extensions*, Figure 8.2) is 2.5× faster than MRI running the pure Ruby version of the benchmarks without the C extensions. JRuby uses JNI to access the C extensions from Java, which causes a significant overhead. Hence it can only achieve 25% of the *MRI With C Extension* performance. JRuby failed to run one benchmark with an error about an incomplete feature. As with Rubinius, 17 of the benchmarks did not make progress in reasonable time. Despite a 8GB maximum heap, which is extremely generous for the problems sizes, some benchmarks in JRuby were spending the majority of their time in GC or were running out of heap.

When running the C extension version of the benchmarks on top of our system the performance is over 32× better than MRI without C extensions and over 3× better than *MRI With C Extension*. When compared to the other alternative implementations of C extensions, we are over 8× faster than Rubinius, and over 12× faster than JRuby, the previous attempt to support C extensions for Ruby on the JVM. We also run all the extensions methods correctly, unlike both JRuby and Rubinius.

In a conventional implementation of C extensions, where the Ruby code runs in a dedicated Ruby VM and the C code is compiled and run natively, the call from one language to another is a barrier that prevents the implementation from performing almost any optimisations. In our system the barrier between C and Ruby is no different to the barrier between one Ruby method and another. We

[1] We reported this issue to the module's authors as `https://github.com/layervault/psd_native/pull/4`.

found that allowing inlining between languages is a key optimisation, as it permits many other advanced optimisations in the Graal compiler. For example, partial escape analysis can trace objects, allocated in one language but consumed in another, and eventually apply scalar replacement to remove the allocation. Other optimisations that benefit from cross language inlining include constant propagation and folding, global value numbering and strength reduction. When disabling cross-language inlining (*JRuby+Truffle With C Extension*) the speedup over MRI is roughly halved, although it is still around 15× faster, which is around 39% faster than *MRI With C Extension*. In this configuration the compiler cannot widen its compilation units across the Ruby and C boundaries, which results in performance that is similar to MRI when running with C extensions.

Figure A.4 shows detailed graphs for all benchmarks and Figure 8.4 shows tabular data. The first column of the tabular shortly describes the application of each benchmark that we evaluate. All other columns show the results of our performance evaluation of the different approaches. We show the absolute average time needed for a single run, the error, and the relative speedup to MRI without C extensions.

The speedup achieved for *MRI With C Extensions* compared to MRI running Ruby code (the topmost bar of each cluster, in red) varies between slightly slower at 0.69x (`psd-blender-compose`) and very significantly faster at 84x (`chunky-operations-compose`).

The speedup of *JRuby+Truffle With C Extensions* compared to MRI varies between 1.37x faster (`chunky-encode-png-image`) up to 101x faster (`psd-compose-exclusion`). When comparing our system to *MRI With C Extensions* we perform best on benchmarks that heavily access Ruby data from C but otherwise do little computation on the C side. These scenarios are well suited for our system because our compiler can easily optimise foreign object accesses and cross-language calls. In some benchmark such as `chunky-colour-r` the entire benchmark, including Ruby and C, will be compiled into a single compact machine code routine. However, if benchmarks do a lot of computations on the C side and exchange little data the performance of our system is similar to *MRI With C extensions*.

If we just consider the contribution of a high performance reimplementation of Ruby and its support for C extensions, then we should compare ourselves against JRuby. In that case our implementation is highly successful at on average over

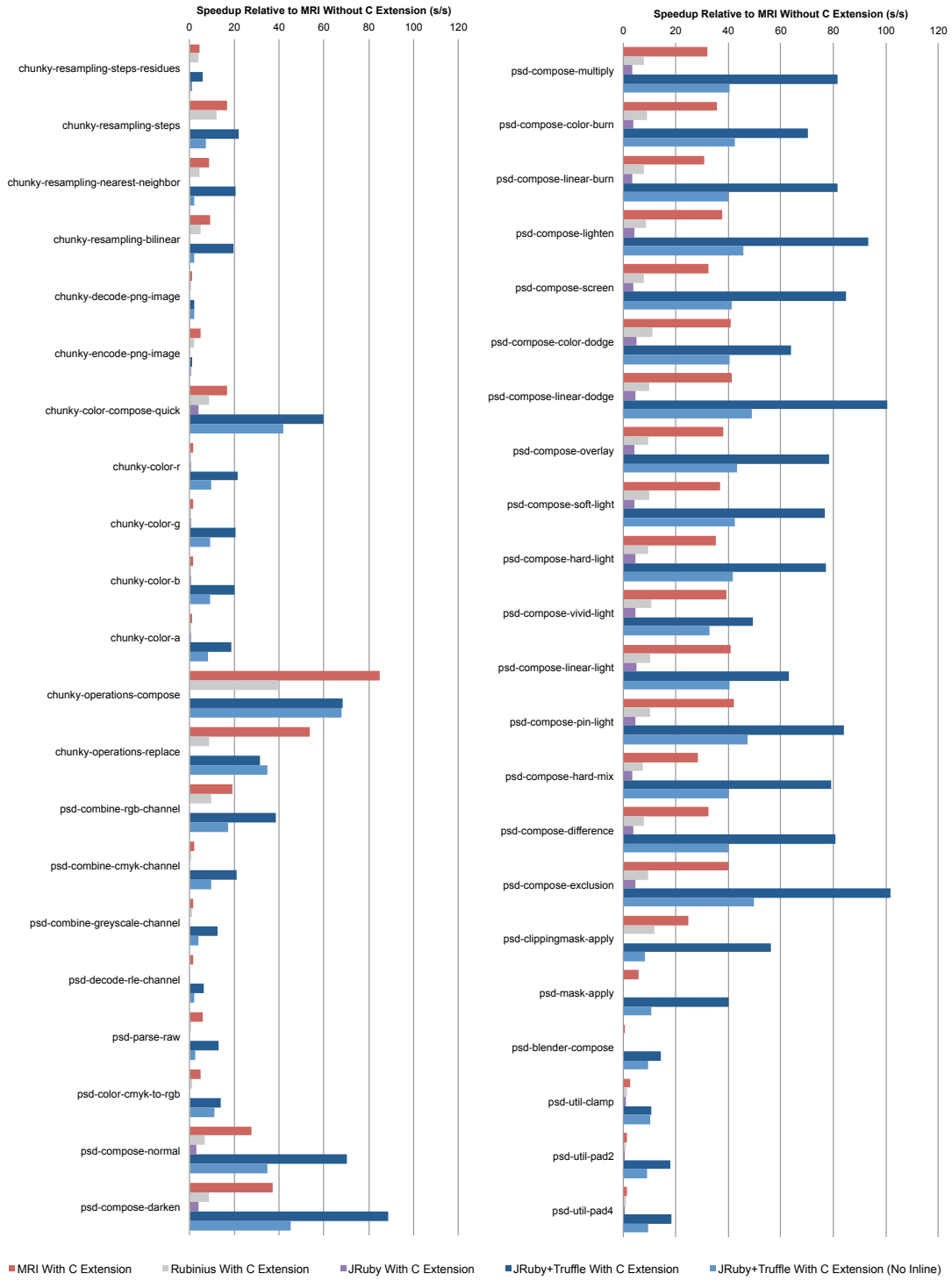


Figure 8.3: Speedup for individual benchmarks.

12× faster. However we also evaluate against MRI directly running native C and find our system to be on average over 3× faster, indicating that our system might be preferable even when it is possible to run the original native code.

8.11 Interfacing To Native Libraries

TruffleC is able to call into native libraries using the *Graal native function interface* [45], which is an FFI understood by Graal for the purposes of optimisation. Given a conventional header file declaration of a function, TruffleC is able to transparently link the function from the current processes symbol table, or with a reference also from an external shared library.

This technique should be enough to interface with libraries such as database drivers, but we have not evaluated in depth. Listing 8.4 shows an example in the Ruby interactive shell, IRB. The system header files `<unistd.h>` and `<stdio.h>` are loaded from disk. These are not special versions of TruffleC, instead we are simply loading the standard headers. We then use a JRuby+Truffle feature that allows us to write C code inline, where we call `printf` with the result from `getpid`. Both of these functions are called natively - they haven't been reimplementd in Java.

```
1  > bin/jruby -X+T bin/irb
2  irb(main):001:0> Truffle::CExt.inline %s{
3  irb(main):002:0*     #include <unistd.h>
4  irb(main):003:0*     #include <stdio.h>
5  irb(main):004:0* }, %s{
6  irb(main):005:0*     printf("Hello, World! I'm %d\n", getpid());
7  irb(main):006:0* }
8  Hello, World! I'm 36641
9  => true
10 irb(main):002:0> exit
```

Listing 8.4: IRB session using TruffleC inline to call a library routine.

8.12 Summary

We have presented a novel solution to the major problem of supporting legacy C extensions in optimised implementations of dynamic languages. We combined

Benchmark	Application	MRI Without C Extesnion			MRI With C Extension			Rubinius With C Extension			JRuby With C Extension			JRuby+Truffle With C Extension			JRuby+Truffle With C Extension (No Inline)		
		Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)	Mean Time (s)	Error (± s)	Speedup (s/s)
chunky-resampling-steps-residues	Calculate offsets for scaling an image	4.1615	0.0006	1.0000	0.8884	0.0003	4.6844	1.0955	0.0013	3.7986	failed	failed		0.6809	0.0075	6.1113	2.9943	0.0158	1.3898
chunky-resampling-steps	Calculate just the integer offsets for scaling	3.1536	0.0032	1.0000	0.1896	0.0003	16.6289	0.2635	0.0043	11.9675	failed	failed		0.1445	0.0002	21.8320	0.4222	0.0002	7.4695
chunky-resampling-nearest-neighbor	Scale an image with no interpolation	4.8069	0.0644	1.0000	0.5449	0.0001	8.8212	1.0989	0.0030	4.3744	failed	failed		0.2354	0.0022	20.4203	2.3568	0.0303	2.0396
chunky-resampling-bilinear	Scale an image with bilinear interpolation	4.9273	0.0262	1.0000	0.5445	0.0003	9.0496	1.0205	0.0103	4.8282	failed	failed		0.2513	0.0038	19.6074	2.3411	0.0295	2.1047
chunky-decode-png-image	Decode a PNG image stream to pixel values	31.9720	0.3122	1.0000	32.0908	0.3098	0.9963	100.3601	2.3741	0.3186	failed	failed		13.7589	0.0101	2.3237	15.7590	0.0120	2.0288
chunky-encode-png-image	Encode pixel values to a PNG image stream	48.0001	0.5826	1.0000	10.0598	0.0503	4.7715	21.8336	0.0560	2.1985	failed	failed		34.8494	0.4705	1.3774	57.5937	0.4539	0.8334
chunky-color-compose-quick	Compose one colour value on top of another	20.0624	0.0049	1.0000	1.2092	0.0017	16.5917	2.2468	0.0022	8.9293	5.2565	0.0749	3.8166	0.3366	0.0003	59.6119	0.4796	0.0028	41.8315
chunky-color-r	Extract colour channel from packed 32bit pixel value	16.0452	0.0205	1.0000	11.2803	0.0221	1.4224	18.1816	0.1635	0.8825	25.5462	0.1627	0.6281	0.7479	0.0003	21.4537	1.6514	0.0133	9.7164
chunky-color-g	Extract colour channel from packed 32bit pixel value	15.1524	0.0397	1.0000	10.6781	0.0188	1.4190	18.4940	0.0289	0.8193	25.2880	0.2154	0.5992	0.7403	0.0006	20.4679	1.6792	0.0222	9.0238
chunky-color-b	Extract colour channel from packed 32bit pixel value	15.0363	0.0445	1.0000	10.1860	0.0042	1.4762	17.7032	0.0409	0.8494	24.7702	0.1849	0.6070	0.7503	0.0051	20.0391	1.6476	0.0234	9.1265
chunky-color-a	Extract colour channel from packed 32bit pixel value	13.8268	0.0514	1.0000	10.0445	0.0244	1.3766	17.9353	0.0412	0.7709	24.6747	0.1415	0.5604	0.7377	0.0006	18.7418	1.6336	0.0144	8.4640
chunky-operations-compose	Compose one image on top of another	18.2619	0.0181	1.0000	0.2156	0.0000	84.7060	0.4620	0.0003	39.5269	failed	failed		0.2682	0.0018	68.1033	0.2689	0.0014	67.9133
chunky-operations-replace	Replace the contents of one image with another	7.5622	0.0063	1.0000	0.1411	0.0000	53.5976	0.8628	0.0004	8.7651	failed	failed		0.2409	0.0067	31.3914	0.2182	0.0045	34.6492
psd-combine-rgb-channel	Decode RGB channel data into pixel values	9.1537	0.0268	1.0000	0.4760	0.0002	19.2303	0.9657	0.0099	9.4787	failed	failed		0.2366	0.0014	38.6804	0.5347	0.0005	17.1209
psd-combine-cmyk-channel	Decode CMYK channel data into pixel values	8.7897	0.0080	1.0000	4.2160	0.0035	2.0848	11.3257	0.1693	0.7761	failed	failed		0.4161	0.0030	21.1240	0.9220	0.0153	9.5333
psd-combine-greyscale-channel	Decode greyscale channel data into pixel values	3.8683	0.0014	1.0000	2.1293	0.0003	1.8167	2.8351	0.0004	1.3645	failed	failed		0.3079	0.0035	12.5636	0.9681	0.0004	3.9960
psd-decode-rle-channel	Decode run-length-encoded data	3.6114	0.0474	1.0000	2.4145	0.0137	1.4957	failed			failed	failed		0.5744	0.0029	6.2866	1.7794	0.0257	2.0296
psd-parse-raw	Decode raw encoded data	3.1876	0.0590	1.0000	0.5426	0.0054	5.8748	5.5173	0.0385	0.5777	failed	failed		0.2460	0.0018	12.9551	1.3409	0.0172	2.3773
psd-color-cmyk-to-rgb	Convert from CMYK to RGB	25.4834	0.0056	1.0000	5.2850	0.0040	4.8218	24.4951	0.0086	1.0403	failed	failed		1.8211	0.0107	13.9934	2.2905	0.0221	11.1257
psd-compose-normal	Compose two pixel values	9.5798	0.0017	1.0000	0.3490	0.0024	27.4506	1.4110	0.0055	6.7895	3.1666	0.0602	3.0253	0.1370	0.0004	69.9514	0.2765	0.0008	34.6531
psd-compose-darken	Compose two pixel values using a filter	12.4882	0.0021	1.0000	0.3358	0.0003	37.1902	1.4342	0.0015	8.7074	3.2332	0.0598	3.8625	0.1408	0.0003	88.6946	0.2770	0.0019	45.0837
psd-compose-multiply	Compose two pixel values using a filter	11.3139	0.0015	1.0000	0.3530	0.0021	32.0465	1.4093	0.0013	8.0278	3.2066	0.0541	3.5284	0.1390	0.0015	81.4240	0.2796	0.0006	40.4717
psd-compose-color-burn	Compose two pixel values using a filter	13.6794	0.0028	1.0000	0.3846	0.0007	35.5709	1.4947	0.0008	9.1519	3.5064	0.0574	3.9013	0.1951	0.0012	70.1328	0.3208	0.0003	42.6349
psd-compose-linear-burn	Compose two pixel values using a filter	11.2797	0.0020	1.0000	0.3664	0.0011	30.7812	1.4525	0.0009	7.7659	3.2223	0.0476	3.5005	0.1385	0.0003	81.4716	0.2822	0.0037	39.9707
psd-compose-lighten	Compose two pixel values using a filter	12.7023	0.0023	1.0000	0.3389	0.0004	37.4836	1.4511	0.0032	8.7538	3.1088	0.0444	4.0860	0.1360	0.0017	93.3989	0.2780	0.0027	45.6998
psd-compose-screen	Compose two pixel values using a filter	11.3777	0.0011	1.0000	0.3491	0.0012	32.5957	1.4418	0.0028	7.8913	2.9852	0.0430	3.8113	0.1344	0.0015	84.6555	0.2761	0.0008	41.2086
psd-compose-color-dodge	Compose two pixel values using a filter	15.6079	0.0031	1.0000	0.3809	0.0007	40.9720	1.4253	0.0018	10.9508	3.0824	0.0449	5.0635	0.2448	0.0014	63.7576	0.3856	0.0031	40.4821
psd-compose-linear-dodge	Compose two pixel values using a filter	13.9772	0.0024	1.0000	0.3381	0.0012	41.3362	1.3973	0.0011	10.0031	3.0626	0.0606	4.5639	0.1389	0.0010	100.6281	0.2850	0.0061	49.0516
psd-compose-overlay	Compose two pixel values using a filter	13.7380	0.0025	1.0000	0.3597	0.0007	38.1892	1.4684	0.0004	9.3555	3.2616	0.0351	4.2120	0.1755	0.0013	78.2569	0.3188	0.0040	43.0996
psd-compose-soft-light	Compose two pixel values using a filter	14.0483	0.0018	1.0000	0.3806	0.0003	36.9104	1.4558	0.0014	9.6496	3.2347	0.0554	4.3431	0.1829	0.0004	76.8086	0.3296	0.0037	42.6223
psd-compose-hard-light	Compose two pixel values using a filter	13.2856	0.0020	1.0000	0.3765	0.0002	35.2864	1.4366	0.0034	9.2477	2.8690	0.0455	4.6307	0.1725	0.0003	77.0179	0.3191	0.0011	41.6411
psd-compose-vivid-light	Compose two pixel values using a filter	15.7572	0.0020	1.0000	0.4015	0.0014	39.2481	1.4759	0.0008	10.6764	3.3599	0.0491	4.6898	0.3199	0.0004	49.2566	0.4782	0.0057	32.9510
psd-compose-linear-light	Compose two pixel values using a filter	15.2815	0.0031	1.0000	0.3725	0.0002	41.0285	1.5195	0.0020	10.0569	3.1666	0.0423	4.8258	0.2426	0.0003	62.9775	0.3796	0.0010	40.2621
psd-compose-pin-light	Compose two pixel values using a filter	15.1247	0.0016	1.0000	0.3601	0.0002	42.0034	1.4739	0.0010	10.2616	3.1750	0.0439	4.7636	0.1800	0.0026	84.0026	0.3203	0.0003	47.2277
psd-compose-hard-mix	Compose two pixel values using a filter	11.2739	0.0022	1.0000	0.3966	0.0012	28.4271	1.5099	0.0017	7.4667	3.1402	0.0598	3.5902	0.1426	0.0003	79.0594	0.2816	0.0033	40.0422
psd-compose-difference	Compose two pixel values using a filter	11.0951	0.0025	1.0000	0.3424	0.0001	32.4050	1.4161	0.0019	7.8352	2.9586	0.0324	3.7501	0.1372	0.0003	80.8976	0.2759	0.0019	40.2215
psd-compose-exclusion	Compose two pixel values using a filter	14.0561	0.0016	1.0000	0.3503	0.0007	40.1213	1.4592	0.0029	9.6328	3.0012	0.0400	4.6836	0.1383	0.0003	101.6350	0.2833	0.0038	49.6157
psd-clippingmask-apply	Apply a clipping mask to an image	6.5653	0.0031	1.0000	0.2642	0.0004	24.8485	0.5486	0.0006	11.9680	failed	failed		0.1169	0.0002	56.1856	0.7980	0.0003	8.2272
psd-mask-apply	Apply an image mask to an image	10.3243	0.0192	1.0000	1.7499	0.0016	5.9001	failed			failed	failed		0.2582	0.0033	39.9857	0.9720	0.0214	10.6212
psd-blender-compose	Blends two images using one of the compose modes	12.7997	0.0762	1.0000	18.6183	0.0019	0.6875	failed			failed	failed		0.9081	0.0040	14.0959	1.3786	0.0199	9.2849
psd-util-clamp	Return a value or minimum or maximum bounds	27.3576	0.0123	1.0000	10.6073	0.0248	2.5791	20.0002	0.0430	1.3679	25.8236	0.2004	1.0594	2.5784	0.0043	10.6105	2.6979	0.0228	10.1401
psd-util-pad2	Round an integer to a multiple of 2	14.4757	0.0443	1.0000	9.7852	0.0098	1.4793	17.8647	0.0140	0.8103	22.6366	0.1362	0.6395	0.8032	0.0004	18.0237	1.6267	0.0141	8.8991
psd-util-pad4	Round an integer to a multiple of 4	15.1029	0.0098	1.0000	10.2633	0.0067	1.4715	17.8847	0.0408	0.8445	23.9633	0.1667	0.6303	0.8314	0.0005	18.1656	1.6326	0.0085	9.2508

Figure 8.4: Description of benchmarks and evaluation data.

two self-optimising AST interpreters, one for C and one for Ruby, and provided an implementation of the MRI API that abstracts over our optimised implementation of language internals.

We could not only demonstrate that our system allows existing C code from production gems can be run in combination with our optimisations, but also that doing so improves performance on actually running the native code by 3×. We therefore both solve the original problem of compatibility with existing C extensions and a new optimising implementation of Ruby, but also improve on performance of the C extensions at the same time.

Chapter 9

Conclusions

In this thesis we have presented JRuby+Truffle, an implementation of the Ruby programming language that out-performs existing implementations of Ruby by an order of magnitude on a set of both synthetic benchmarks, and a set of benchmarks from real-world production libraries which exercise metaprogramming patterns that are common in the Ruby ecosystem.

JRuby+Truffle optimises for complex metaprogramming patterns that Ruby programmers use to reduce code duplication and add abstraction to simplify programs. Existing implementations of Ruby do not optimise for these patterns, and there previously existed no techniques to do so. We have introduced dispatch chains as a generalised form of polymorphic inline caches to optimise a range of metaprogramming features in the Ruby language, and shown the impact that this has on benchmarks created from production libraries.

JRuby+Truffle does not achieve high performance by limiting the features of the Ruby language that it supports to those which are easy to implement and do not conflict with the optimisations we have used. In fact, it supports around 93% of the Ruby language and around 87% of the core library. It also does not achieve high performance by putting difficult features of the language behind debug flags which are disabled by default, and drastically limit performance if they are enabled. In JRuby+Truffle there are no language feature flags, and even complex language features such as `ObjectSpace.each_object` are always enabled. No other implementation of Ruby achieves these two things at the same time. We have introduced the new technique of guest-language safepoints to do this.

Tooling is also not compromised in our implementation. JRuby+Truffle features debugging support which is always enabled and can be used on a long running process and will not damage performance. Utilising the new concept of wrapper nodes for dynamic instrumentation of an optimised AST, breakpoints can be set in optimised code with no overhead, and conditional breakpoints only incur cost proportional to the cost of evaluating the condition.

JRuby+Truffle does not discard support for legacy applications. The workaround for low performance of existing implementations of Ruby is to write C extensions which bypass much of the dynamic nature of Ruby and allow native primitives to be used instead. JRuby+Truffle can run C extensions by interpreting them using the same dynamic techniques as it uses to run Ruby code. The abstraction of interpretation allows optimisations made to runtime and application data structures while still providing the illusion of a native interface to the C code. We believe that JRuby+Truffle is the first high performance re-implementation of a dynamic programming language that also provides a high performance implementation of C extensions.

JRuby+Truffle is a demonstration that an extremely large and complex industrial language such as Ruby can be optimised for the way that programmers want to use it, without limiting the features that are made available, and with high performance tooling support. Although we have developed sophisticated new techniques, the implementation of JRuby+Truffle itself is not complex and many of our techniques are now being pushed down into the underlying Truffle language implementation framework where they are being used by other language implementations. In fact, JRuby+Truffle is a whole level of abstraction simpler than any other implementation of Ruby, as it is an AST interpreter, and does not include any bytecode interpretation or code generation phase.

Appendix A

Dataflow and Transactions for Dynamic Parallelism

A.1 Introduction

This thesis has covered techniques for the implementation of dynamic languages with optimisations that work for metaprogramming operations and are compatible with debug tooling. This appendix considers a problem that is related and may be a basis for future work in the JRuby+Truffle project, but is not at the moment implemented as part of that project or using the Ruby language.

In this appendix we extend the group of concepts we are considering under the dynamic umbrella to include *dynamic parallelism*. In the same way that it is sometimes desirable to defer actions such as method binding until runtime because it gives more flexibility, it can also be more flexible to allow a system of parallel tasks to be defined at runtime. In particular we are interested in *irregular parallelism*. This is the class of applications that we may wish to parallelise, where the tasks to be executed and the dependencies between them may not be statically determinable. In extreme cases, as is the case with the workload we have studied in this appendix, dependencies between tasks may not be determinable without knowing the results of the tasks.

In the past this has been solved with pessimistic locking schemes, which are difficult to get right. Instead we can apply an *optimistic* system that speculates that most tasks will not conflict, and works under that assumption until a problem is found. There is a parallel between this and the speculative optimisation

that we have applied to solve metaprogramming and tooling problems in a high-performance implementation of Ruby. There is also a parallel to the work in Chapter 7 in the discussion of applications such as reverting biased locks and aborting failed speculative parallelism–such techniques could have been applied to the work in this appendix.

The work in this Appendix applied an existing framework for dataflow [40] and transactional memory [39] in the Scala programming language The independent research contribution was to produce a novel technique to solve a representative irregular parallel application [12].

A.2 Irregular Parallelism

A great many strategies have been proposed to make writing parallel programs that run on multicore shared-memory systems easier and less error prone, at the same time as achieving a good return on the invested number of processors. This endeavour becomes more pressing as multicore systems become the majority on servers, desktops and mobile devices. The number of cores looks likely to continue to increase, requiring more parallelism in our programs in order to exploit this power.

We have looked at work to combine two existing constructs that have shown considerable potential on their own – dataflow [110] and transactional memory [50]. We have used an established benchmark, Lee's algorithm for routing printed circuit boards, to make an early assessment of their utility for creating efficient, simply written and correct parallel programs. This paper shows how using the DFLib and MUTS [39] implementations of dataflow and transactional memory makes the parallel implementation of our program simpler, at the same time as achieving a real world performance increase compared to coarse locks on typical desktop hardware, even when all overhead is included.

A.3 Dataflow

Dataflow decomposes a program into a directed acyclic graph with nodes that perform computation and edges that take the output of one computation and provide it as input to one or more other nodes. A node is runnable if all of the nodes preceding it have finished their computation, and so all of the inputs are

available. In a model where the computations do not have any side effects, such as in functional languages, the graph completely expresses dependencies between nodes, and if more than one node is runnable at the same time then they may be run in parallel. The name *dataflow* emphasises that it is the data flowing between nodes that causes them to be scheduled to run, rather than the a program counter reaching procedures and causing them to run, as in the von Neumann model.

As well as helping to expose parallelism by dividing a program into groups of computations without interdependencies, dataflow can also help us by handling the synchronisation between running threads. Threads will only run when all of their inputs are already available, so there is no code needed to achieve the common barrier or join operations as in Java threads. However, dataflow does not help us to address the problem of a shared mutable state. As we shall show using our example problem, Lee's algorithm as described in Section A.5, shared state can be a key part of the algorithm.

For this work we used the Scala dataflow library, DFLib, being developed as part of the Teraflux project at the University of Manchester. A dataflow node is created by wrapping a Scala function in a `DFThread` object that has methods to either explicitly set an input, or to link an input to the result of another `DFThread`. This allows a dataflow graph to be built up and implicitly run by DFLib, which will schedule runnable functions using enough OS threads to occupy all hardware threads.

A.4 Transactions

Transactional memory allows a series of memory reads and writes to be executed as if they were a single indivisible, or atomic, operation. This removes the need for explicit mutually exclusive locks around data structures. Instead, a data structure is modified within an atomic block and so appears to be a single operation that can be applied without excluding others. There are many different algorithms that implement this basic transactional behaviour [50], but this programming interface is common to most of them.

Locks can be a blunt tool that disallows concurrent access to memory based on the assumption that such accesses will conflict. For some applications, this may be a correct assumption, but given a particular problem we may be confident that conflict is rare. Most implementations of transactional memory allow transactions

that do not have conflicting memory accesses to proceed in parallel, dealing with the less common case of conflicting memory access by restarting one or both of the transactions involved. This is in contrast to a standard locking approach which would disallow parallel memory accesses on the same data structure, even when they do not conflict.

For the work presented here we used the Manchester University Transactions for Scala (MUTS) library. Taking the Java Deuce transactional memory library [62] as a starting point, MUTS extends and modifies this work to implement a selection of different techniques for implementing software transactional memory for the Scala programming language.

MUTS uses a Java agent to visit all Java classes as they are loaded into the JVM. This allows it to create a transactional version of all methods that instrument read and write operations. MUTS can then pass these reads and writes to one of several implementations of software transactional memory algorithms included in the library. A typical algorithm stores reads and writes in a log, deferring writing to shared memory from the log until the transaction is complete and values in the read log are verified to not have been written to by another thread in the elapsed time.

A.5 Lee's Algorithm

Lee's algorithm [66] solves the problem of finding independent routes between a set of pairs of points on a discrete grid. The applications originally proposed included drawing diagrams, wiring and optimal route finding, but the algorithm is now best known as a method for routing paths on a printed circuit board. There are later algorithms for route finding with less computational complexity, but Lee produces a shortest solution for any given route and board state, as opposed to using heuristics to arrive at a good-enough solution in less time. Figure A.2 shows the output of one application of Lee's algorithm – routing paths on a printed circuit board – as generated by our implementation.

A simple overview of the Lee algorithm is that from the start point of the route it looks at all adjacent points and marks them as having cost one. From each point P so marked, it marks each adjacent point P_{adj} as having cost $cost(P_{adj}) = cost(P)+1$, as long they were not already marked with a lower cost. If an adjacent point is already part of another route then using that point would cost more, by

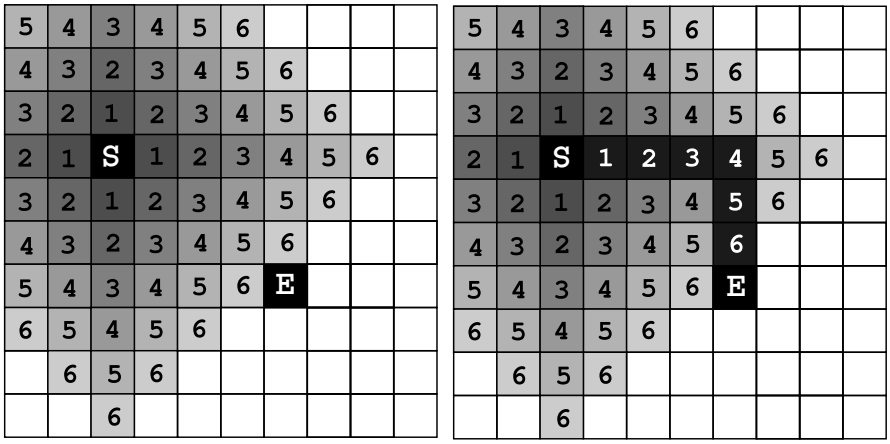


Figure A.1: Expand and trace phases of Lee's algorithm[12]

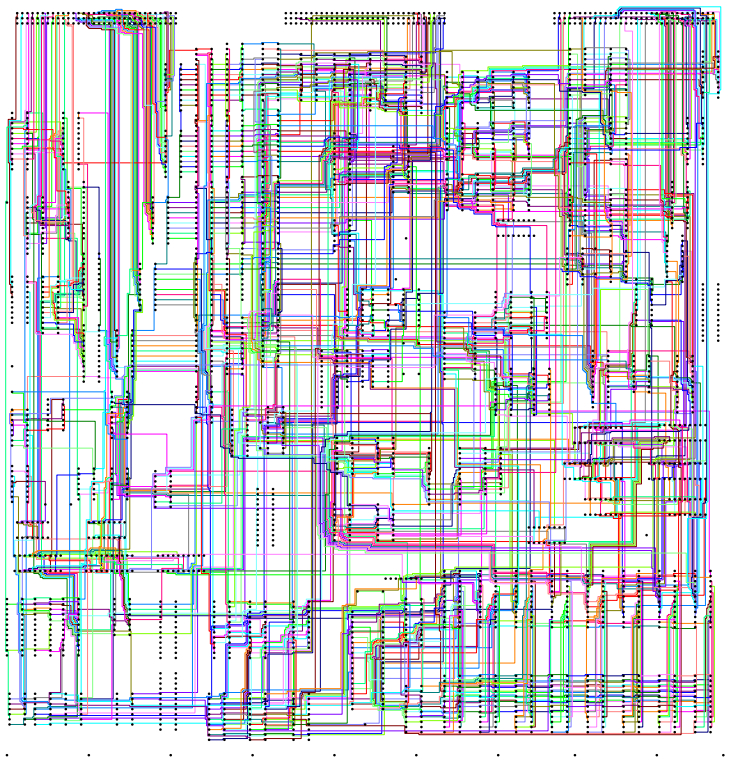


Figure A.2: A solution from Lee's algorithm

an arbitrary constant factor, as a bridge of one route over another needs to be built. This expansion, as it is called, continues until the end point is a member of the set of adjacent points. Typically, this expansion forms a circle around the start point, enveloping obstacles such as existing routes, and finishing at the end point.

A trace is then run from the end point to the start point, always moving to a point of lower cost until the start point is reached. This produces a route which can be marked on the grid. These two key steps are illustrated in Figure A.1.

Unlike classic problems such as finding Fibonacci numbers or merge sort, Lee does not belong to the class informally known as *embarrassingly parallel*. Such problems can be broken down into smaller problems that are entirely independent, and so can be executed in parallel. There are several ways to decompose Lee's algorithm into a set of subproblems, but it cannot be guaranteed a priori that any two subproblems will not want to lay a route in the same cell of the shared grid, even when the route's start and end points are known.

The key to the problem is that any set of subproblems still need to share access to a single resource – the grid. It is not simple for each thread to have an independent copy of the grid, as two threads could need use the same point for multiple routes and would then have to synchronise between themselves. This would add logic to the program that is unrelated to the algorithm that we are implementing. It is also not simple for each thread to have one part of the larger grid, as you can not guarantee which parts a route will use before the expansion has been calculated and such a scheme would vastly increase the complexity of the program. However, given all routes on a circuit board it is unlikely that any two being routed at a particular time will conflict. The parallelism is there – it is just that it is hard to determine statically and is more apparent as the program is running.

Work has already been done to use Lee to evaluate the runtime characteristics of a transactional program [109], and to evaluate the performance of implementations of transactional memory for Java [12]. Our work builds on this by evaluating the combined use of transactions and dataflow, with Scala, MUTS and DFLib.

A.6 Implementation

Scala [80] is a hybrid functional and imperative programming language that runs on the Java Virtual Machine. The functional aspect allows a clean expression of a problem in a way that is open to parallelisation with minimum shared or mutable state. The imperative aspect allows us to have controlled mutation of the shared state when we require it for efficient execution.

We wrote a set of programs in Scala to solve Lee. One sequential, *seq*; one using coarse locks, *coaselock*; one using the MUTS library, *muts* and one using both MUTS and DFLib, *dataflow*. They share common input and output formats and many internal data structures. The output of the parallel implementations is non-deterministic due to the interleaving of separate threads of execution, so we check the the total routing cost of the solution to compare the equivalence of different implementations.

A.6.1 Sequential

We first created a sequential implementation of Lee using a purely functional approach, combined with a central mutable data structure to represent the board. Our implementation of Lee is minimal and we excluded some refinements that Lee described such as weighting against turns in paths. While these refinements are sensible for actual routing applications, they do not have any effect on the parallel characteristics of the program and can be considered constant factors to performance in both space and time. We also allow only one level of bridging, where one route can cross an existing route, as this is usually sufficient for our test boards.

The sequential program, *seq*, represents a clear and succinct expression of the algorithm. We are adding parallelism because we want the program to run faster, not because the algorithm requires it, so ideally we want this optimisation to require minimum new code and minimum modifications to existing code. The perfect expression of parallelism would be completely orthogonal to the sequential expression of the algorithm.

A.6.2 Coarse Lock

Our first parallel implementation, *coarselock*, creates a thread for each hardware thread in the system. Each thread works in a loop. First it allocates a route from the input and obtains a private copy of the shared board data structure. Then it expands and traces the route, using this private copy. The route is then validated, to ensure that since the private copy was made the board has not been changed by another thread to make the proposed solution invalid and then commits the route to the shared board data structure.

There are three resources being shared here – the board data structure, the list of available routes and the list of solutions. Additionally, the master thread needs to know when all the solutions are in the list, which we achieve by waiting for all threads to finish with the `join()` method.

We make access to these shared resources mutually exclusive using Scala's implementation of the M-structure [15], `SyncVar[T]`. For example, an instance of `SyncVar[MutableBoardState]` holds the shared mutable board state data structure. When a thread wants to lock the data structure so it can validate its route and commit it to the board without interruption by another thread, it calls the `take` method to atomically remove the data structure and leave the `SyncVar` empty. Any other thread trying to read or write from the same data structure will block within `take` until the former thread is done with the data structure and calls the `put` method to return it for other threads to use.

```
1  val boardStateForFreeze = boardStateVar.take()          // Lock
2  val privateBoardState = boardStateForFreeze.freeze
3  boardStateVar.put(boardStateForFreeze)                  // Unlock
4
5  val expansion = expandRoute(board, route, privateBoardState)
6  val solution = traceRoute(board, route, expansion)
7
8  val boardStateForLay = boardStateVar.take()             // Lock
9  val verified = verifyRoute(route, solution, boardStateForLay)
10 if (verified)
11     layRoute(route, solution, boardStateForLay)
12 else
13     scheduleForRetry(route)
14 boardStateVar.put(boardStateForLay)                     // Unlock
```

Listing A.1: Parallel Lee's algorithm using a global `SyncVar`

In this implementation we had a single lock for the entire board. Another option would be to use multiple locks to control different parts of the board. We haven't created such an implementation, but we do refer to this strategy in the analysis section.

A.6.3 MUTS

We used the MUTS library to create a parallel implementation by modifying *coarselock*. Where *coarselock* acquires a resource with `take` to the exclusion of all other threads before `put`ting it back when it is done, we can instead perform that action inside a transaction that will only allow other threads to read the same values as long as they don't write to them, and will automatically retry if such a conflict is found.

```
1  // Atomically copy the shared data structure
2  val privateBoardState = atomic { boardState.freeze }
3
4  val expansion = expandRoute(board, route, privateBoardState)
5  val solution = traceRoute(board, route, expansion)
6
7  // Atomically write to the shared data structure
8  atomic {
9      if (verifyRoute(route, solution, boardState))
10         layRoute(route, solution, boardState)
11     else
12         scheduleForRetry(route)
13 }
```

Listing A.2: Parallel Lee's algorithm using transactional memory

This code looks very similar to the use of a `pthread\_mutex\_t`, a Java `synchronized` block or a `SyncVar` as used in *coarselock*. The idea that `atomic` could be implemented as a global single lock is one model for thinking about transactional memory [50]. However, in practice the `atomic` blocks will allow multiple threads to read at the same time, and to write at the same time as long as they do not try to write routes using the same point. If there is a conflict, they will be retried, just as *coarselock* does explicitly.

A.6.4 Dataflow

We then extended our *muts* implementation to use the Scala dataflow library, DFLib. Where the *muts* implementation created parallelism by spawning Java threads, the dataflow constructs provided by DFLib allow us to express this creation of parallelism in a different way. Each route is a `DFThread` that will have its inputs ready at the start of the program's execution and so will all be runnable. DFLib will schedule them for us so that only a sensible number are running at any time.

A final `DFThread`, the *collector thread* will be then created that has each route's `DFThread` as one of its arguments. This will therefore be run when the solutions are complete. This is a convenience construct provided by DFLib, as it is expected to be a common pattern, and replaces the synchronisation needed to create the list of solutions and the `join` operation to wait for all threads to finish that we used in *coarselock*. Figure A.3 shows the resulting dataflow graph, and illustrates how the scheduler executes a small subset of routes at time.

```
1  // Accepts solutions as arguments and build a list from them
2  val solutionCollector = DFManager.
3      createCollectorThread[Solution](routes.length)
4
5  for (route <- routes) {
6      // Create a thread to solve a route
7      val routeSolver = DFManager.createThread(solveRoute _)
8
9      routeSolver.arg1 = board
10     routeSolver.arg2 = route
11     routeSolver.arg3 = boardState
12
13     // It will send the solutions to this function
14     routeSolver.arg4 = solutionCollector.token1
15 }
```

Listing A.3: Parallel Lee's algorithm using DFScala

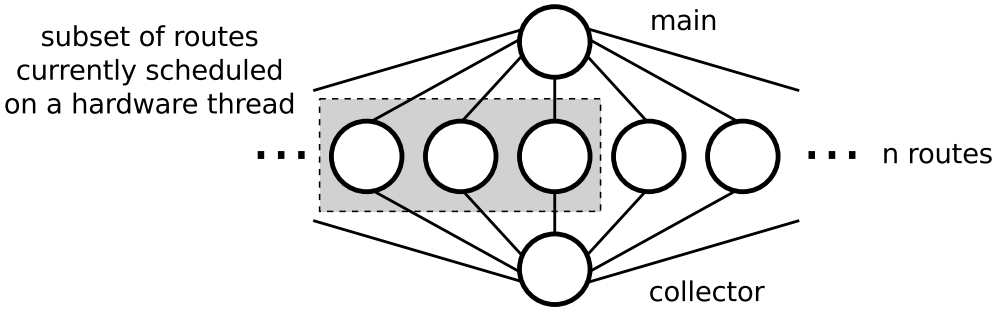


Figure A.3: Example dataflow graph for Lee's algorithm

A.7 Analysis

A.7.1 Methodology

The programs were compiled with Scala 2.9.1 and run on the JVM 1.6.0_27 on an Intel Core i7 processor comprising four physical cores, each with two-way hyper-threading for eight hardware threads. The OS was openSUSE 11.2 with Linux 2.6.31.14.

A.7.2 Results

Table A.1 shows the running time of the core part of each program, measured using `System.nanoTime()`. This excludes the startup time for the JVM and in the case of *muts* and *dataflow*, time to rewrite classes for transactional access. Each program was run ten times with mean average and standard deviation taken, rounded to three decimal places. programs were constrained to use 1, 2, 4, 6 or all 8 of the available hardware threads by replacing calls to `availableProcessors()` with a constant value. These results indicate the relative performance than can be achieved within a longer running system. All parallel implementations show a decrease in performance when using all available hardware threads. This is in line with other researchers' findings [72] and is probably caused by higher contention on resources, limits of hyper-threading, and time sharing with the kernel and other system processes. We show these results as they are what would be achieved with a simple implementation that would by default try to use all available hardware threads.

Figure A.4 shows the resulting speedup of the algorithms, compared to the sequential implementation. That is the time for the sequential implementation divided by time for each other implementation, for a given number of hardware

Impl.	Hardware Threads 1		2		4		6		8	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
seq	29.355	0.310	29.396	0.212	29.535	0.303	29.376	0.220	29.330	0.215
coarselock	30.374	0.321	17.485	0.396	14.986	1.266	13.748	0.776	21.961	1.550
muts	31.422	0.421	16.648	1.157	13.357	0.493	11.528	0.425	14.869	0.683
dataflow	32.093	0.282	16.994	1.149	13.630	1.105	11.805	0.460	14.570	0.401

Table A.1: Mean algorithm running time (seconds)

threads. These results indicate the return on investment for the number of hardware threads applied to the problem.

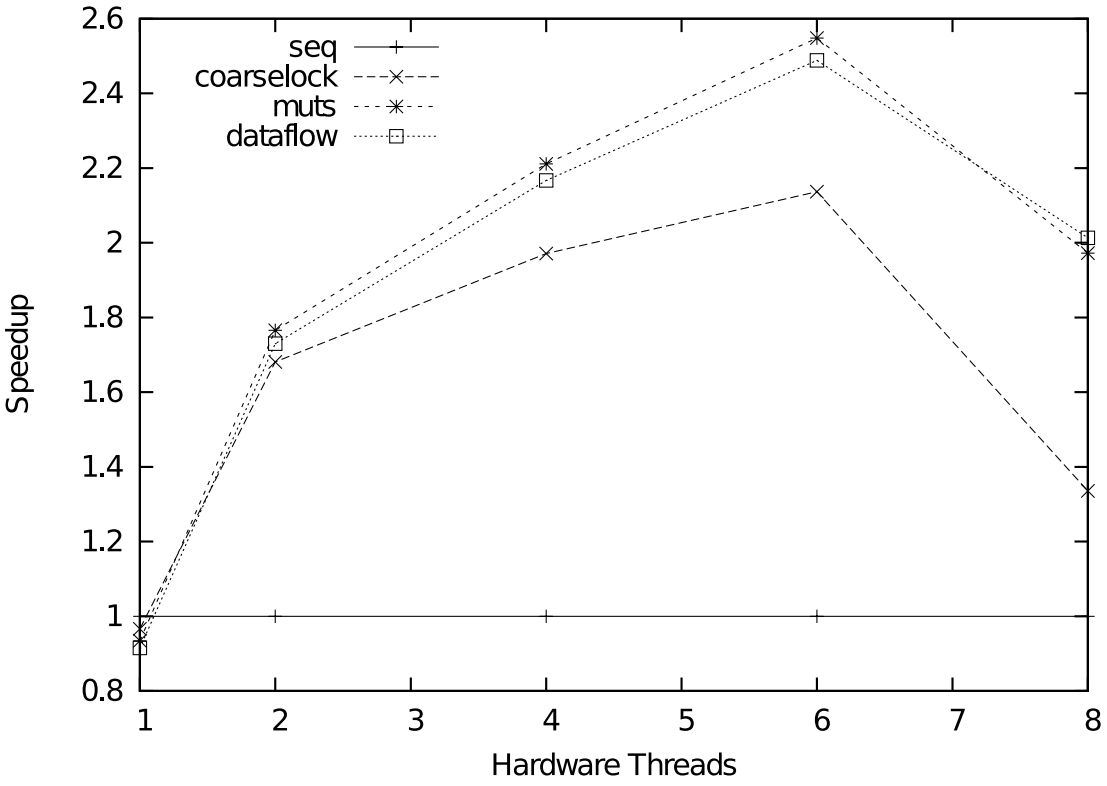


Figure A.4: Algorithm speedup compared to sequential

Table A.2 shows the running time of the entire program shown next to the algorithm time, when running on 8 hardware threads. This includes JVM startup (common to all implementations), and for *muts* and *dataflow* the time to rewrite classes for transactional access. These results are indicative of the total real-world cost of implementing Lee's algorithm for a medium sized board in each of the implementations, including all overhead of the libraries that we have employed,

Implementation	Algorithm	Whole program	Overhead
seq	29.330	29.697	0.367
coarselock	21.961	22.634	0.673
muts	14.774	17.136	2.267
dataflow	14.215	18.877	4.307

Table A.2: Whole program running time on 8 hardware threads (seconds)

Implementation	Lines of code	Parallel operations
seq	251	0
coarselock	330 (+79)	11
muts	328 (+77)	6
dataflow	300 (+49)	5

Table A.3: Code metrics

and the rewriting cost involved in MUTS. Rewriting could be done ahead of time, and we show these numbers here to give an indication of this cost, whether it is made at runtime or not. The overhead becomes a less significant proportion of whole program time given larger problem sizes of a greater number of problems processed in batch.

Table A.3 shows some informal metrics of the program code required to implement the different parallel algorithms. *Parallel operations* refers to the number of operations at the source code level related to nothing but the parallel architecture around the pure algorithm. This includes creating or joining a thread, reading or writing a synchronisation variable and atomic blocks. Each parallel operation detracts from the pure algorithm and is a potential source of error.

All of the parallel implementations are correct and usable, but we had two goals that we can analyse them against. First, the only reason for creating parallelism was that we wanted the program to run faster than our sequential version. We therefore analysed their performance against the sequential implementation, given a multicore system. Secondly, we said that ideally we didn't want to distract from the elegant expression of the algorithm that the sequential implementation gives us. We wanted to introduce minimum new code and to modify minimum existing code. Each addition or modification further ties the expression of the algorithm and the parallelism together and makes modification or debugging to the algorithm itself harder. We therefore analyse the changes needed to create

the different parallel implementations.

A.7.3 Coarse Lock

Implementing Lee's algorithm using coarse locks would probably be the default approach by most industrial programmers. Shared data structures has been identified and locks placed around them. This has created a more verbose program with 11 parallel operations.

The key problem with *coarselock* is that all threads need to read and write the board to make any progress, and given that we are making that access mutually exclusive, we are only allowing one thread to make progress at a time. There is still a degree of work that can be done in parallel – after creating a private copy of the board, the expansion can proceed in parallel – but when we add more threads trying to complete the work faster, we just end up with a bigger queue for the board lock.

As described in Section A.6, it would be possible to develop a finer grained system of locks. However, *coarselock* is already the most complicated of the programs that we have written, as shown in Table A.3, and that is with just one lock. Multiple locks would require logic to work out which locks to acquire, as well as a protocol for the order in which to acquire the locks in order to avoid classic deadlock. Even if it tested well, how would we gain confidence that our locking protocol worked every time?

A.7.4 MUTS

The *muts* implementation looks similar to *coarselock*, with the same thread creation and the same points of synchronisation on the same data structures. However, as we are using transactional memory, the semantics of the code is very different, and will not block another thread unless there is a conflict in the memory locations that they want to read and write. The *muts* implementation achieves better performance from essentially the same code as in *coarselock* because the MUTS `atomic` block will allow more than one thread to run inside it, as long as they are not conflicting on the memory that they use, which as we already described, is unlikely.

Software transactional memory introduces a significant overhead to programs, in that within a transaction all reads and writes have to be instrumented and

logged. This will entail allocating and populating a data structure in proportion to the number of reads and writes the transaction performs. For example, to create a private copy of our test boards involves creating a log with a not-insignificant 600^2 read entries. In MUTS, there is also the overhead at runtime of rewriting all class files to include transactional variants of methods used within transactions. In this evaluation we are looking at the resulting performance of the program, so we included all of these overheads in our whole program timing measurements. This transformation can alternatively be made ahead of time for a known set of class files. Even with all of the overhead, *muts* still runs significantly faster than *seq* and *coarselock* on 8 hardware threads.

A.7.5 Dataflow

The dataflow implementation uses the same code as *muts* to synchronise access to the shared board data structure, but by structuring the program as dataflow we simplify the parallel parts of the algorithm. As we create one `DFThread` for each route, we have removed the synchronisation needed for sharing the available remaining routes between worker threads. By creating a final `DFThread` to collect the results we have also removed the synchronisation needed there. DFLib can manage the scheduling and dependencies of both of these two problems for us. This reduces the number of parallel operations to a lower number than that of *muts* alone. We would argue that where less parallel constructs are required, development is easier and there is less possibility for error, as has been found empirically by other researchers [84].

A.8 Further Work

A.8.1 DFLib

Our implementation of Lee's algorithm uses dataflow to express only a couple of simple dependencies, and although this is a legitimate use of DFLib that does reduce the volume of code needed for synchronisation and a work-queue, it is likely that a more advanced dataflow decomposition of Lee's algorithm, or another problem entirely, will reveal much greater gains in elegant expression and runtime performance available using DFLib.

A.8.2 MUTS

MUTS will currently make any read or write operation within a transactional block part of the transaction, regardless of whether or not the object is shared or mutable. MUTS could be improved with static analysis to reduce the size of the read and write sets. Other transactional memory implementations such as that of Haskell achieve this with explicitly typed transactional objects [51], but if this was the case with MUTS we could not have used our existing sequential board data structure without modification, and modifying it to use a transactional type such as Haskell's `TVar` would have been more work unrelated to the actual algorithm.

A.9 Summary

Our evaluation shows that dataflow combined with transactional memory is a succinct and efficient method for a parallel implementation of Lee's algorithm and is worth further development and investigation.

When applied to Lee's algorithm, dataflow and transactions allow a parallel implementation that is closest to the original sequential implementation. This makes any modifications needed to the algorithm simpler, as one has to consider less parallel code, and it reduces the chance of error as there are less instances of their use that could be incorrect.

These methods expose more parallelism than simple coarse locks and even with runtime transactional overhead the core of the implementation always runs faster than coarse locks, and with 8 hardware threads will run faster even when time consuming rewriting is included in timings.

We believe that transactions and dataflow in Scala using MUTS and DFLib can be used in other research development and real world applications to express parallel programs with minimal modifications and extra code, while achieving good comparative performance and speedup.

Appendix B

Benchmarks

B.1 Synthetic Benchmarks

These are classic research synthetic benchmarks, collected by the Computer Language Benchmarks Game [31] (formerly known as the Computer Language Shootout) and the Topaz project [37].

Benchmark	Exercises
classic-binary-trees	Object allocation
classic-fannkuch-redux	Array access [11]
classic-mandelbrot	Floating point
classic-n-body	Floating point and object access
classic-pidigits	`Bignum`
classic-spectral-norm	Floating point
classic-richards	Polymorphic call sites
classic-deltablue	Logic
classic-red-black	Object allocation
classic-matrix-multiply	Floating point and array access
topaz-neural-net	Floating point, object access, higher order methods

B.2 Production Benchmarks

B.2.1 Chunky PNG

Chunky PNG [107] is a Ruby library for reading and writing PNG image files, managing colour values, simple drawing operations and algorithms such as resampling. We have taken every method where a native equivalent is found in Oily PNG [67] and packaged them as a benchmark.

Benchmark	Description
chunky-canvas-resampling-steps-residues	Integer and floating arithmetic
chunky-canvas-resampling-steps	Integer and floating arithmetic
chunky-canvas-resampling-nearest-neighbor	Integer and floating arithmetic
chunky-canvas-resampling-bilinear	Integer and floating arithmetic
chunky-decode-png-image-pass	Unpacking binary data
chunky-encode-png-image-pass-to-stream	Unpacking binary data
chunky-color-compose-quick	Integer arithmetic and packing
chunky-color-r	Bit manipulation
chunky-color-g	Bit manipulation
chunky-color-b	Bit manipulation
chunky-color-a	Bit manipulation
chunky-operations-compose	Integer arithmetic and packing, array indexing
chunky-operations-replace	Copying between arrays

B.2.2 PSD.rb

PSD.rb [69] is a Ruby library for reading and writing Photoshop Document image files, accessing the graph of objects within them, manipulating font data, and implementations of Photoshop standard composition functions. We have taken every method where a native equivalent is found in PSD Native [68] and packaged them as a benchmark.

Benchmark	Description
psd-imagemode-rgb-combine-rgb-channel	Bit manipulation
psd-imagemode-cmyk-combine-cmyk-channel	Array sorting and indexing
psd-imagemode-greyscale-combine-greyscale-channel	Bit manipulation
psd-imageformat-rle-decode-rle-channel	Array accessing
psd-imageformat-layerraw-parse-raw	Copying an array
psd-color-cmyk-to-rgb	Hash maps, array sorting and index
psd-compose-normal	Integer arithmetic and metaprogramming
psd-compose-darken	Integer arithmetic and metaprogramming
psd-compose-multiply	Integer arithmetic and metaprogramming
psd-compose-color-burn	Integer arithmetic and metaprogramming
psd-compose-linear-burn	Integer arithmetic and metaprogramming
psd-compose-lighten	Integer arithmetic and metaprogramming
psd-compose-screen	Integer arithmetic and metaprogramming
psd-compose-color-dodge	Integer arithmetic and metaprogramming
psd-compose-linear-dodge	Integer arithmetic and metaprogramming
psd-compose-overlay	Integer arithmetic and metaprogramming
psd-compose-soft-light	Integer arithmetic and metaprogramming
psd-compose-hard-light	Integer arithmetic and metaprogramming
psd-compose-vivid-light	Integer arithmetic and metaprogramming
psd-compose-linear-light	Integer arithmetic and metaprogramming
psd-compose-pin-light	Integer arithmetic and metaprogramming
psd-compose-hard-mix	Integer arithmetic and metaprogramming
psd-compose-difference	Integer arithmetic and metaprogramming
psd-compose-exclusion	Integer arithmetic and metaprogramming
psd-renderer-clippingmask-apply	Integer arithmetic and metaprogramming
psd-renderer-mask-apply	Integer arithmetic
psd-renderer-blender-compose	Integer arithmetic and metaprogramming
psd-util-clamp	Array sorting and indexing
psd-util-pad2	Bit manipulation
psd-util-pad4	Bit manipulation

Bibliography

[1] How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Communications of the ACM*, 29, March 1986.

[2] Ruby Summer of Code Wrap-Up. http://blog.bithug.org/2010/11/rsoc, 2011.

[3] Standard ECMA-335. Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm, 2012.

[4] JDK-4836252: allow breakpoints in compiled code, 2013.

[5] jruby-cext: CRuby extension support for JRuby. https://github.com/jruby/jruby-cext, 2013.

[6] Why shouldn't I use PyPy over CPython if PyPy is 6.3 times faster? http://stackoverflow.com/questions/18946662/why-shouldnt-i-use-pypy-over-cpython-if-pypy-is-6-3-times-faster, 2013.

[7] Private correspondence with Tim Felgentreff, 2015.

[8] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi. The HipHop virtual machine. In *OOPSLA '14: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 777–790, New York, New York, USA, Oct. 2014. ACM Request Permissions.

[9] J.-h. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[10] J.-h. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic Inference of Static Types for Ruby. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, 2011.

[11] K. R. Anderson and D. Rettig. Performing Lisp analysis of the FANNKUCH benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, Oct. 1994.

[12] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Lee-tm: A non-trivial benchmark for transactional memory. In *Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processin*, 2008.

[13] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[14] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. of the Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12. ACM, 2000.

[15] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. 1991.

[16] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proc. of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25. ACM, 2009.

[17] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *OOPSLA '13: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 167–182, New York, New York, USA, Oct. 2013. ACM Request Permissions.

[18] C. F. Bolz and L. Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 2013.

[19] S. Brunthaler. Efficient Interpretation Using Quickening. In *Proc. of the Symposium on Dynamic Languages*, number 12 in DLS, pages 1–14. ACM, Oct. 2010.

[20] M. Chevalier-Boisvert and M. Feeley. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 1–24, June 2015.

[21] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, New York, USA, Oct. 1999. ACM Request Permissions.

[22] C. Click and M. Paleczny. A Simple Graph-Based Intermediate Representation. *Intermediate Representations Workshop*, 30(3):35–49, 1995.

[23] B. Daloze, C. Seaton, D. Bonetta, and H. Mössenböck. Techniques and Applications for Guest-Language Safepoints. In *Proceedings of the 10th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS)*, 2015.

[24] M. T. Daly, V. Sazawal, and J. S. Foster. Work In Progress: an Empirical Study of Static Typing in Ruby. In *Proceedings of the PLATEAU Workshop on Evaluation and Usability of Programming Languages and Tools*, 2009.

[25] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System, 1984.

[26] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[27] G. Duboscq, T. Würthinger, and H. Mössenböck. Speculation without regret: reducing deoptimization meta-data in the Graal compiler. In *PPPJ '14: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 187–193, 2014.

[28] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *VMIL '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, 2013.

[29] M. J. Edgar. Static Analysis for Ruby in the Presence of Gradual Typing. Master's thesis, Dartmouth College, 2011.

[30] T. Felgentreff, M. Springer, et al. MagLev, 2014.

[31] B. Fulgham and I. Gouy. The Computer Language Benchmarks Game.

[32] M. Furr, J.-h. An, J. S. Foster, and M. Hicks. The Ruby Intermediate Language. In *Proceedings of the Dynamic Language Symposium*, 2009.

[33] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009.

[34] Y. Futamura. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999, Originally published in 1971.

[35] K. Gade. Twitter Search is Now 3x Faster. 2011.

[36] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2009.

[37] A. Gaynor, T. Felgentreff, et al. Topaz, 2014.

[38] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[39] D. Goodman, B. Khan, S. Khan, C. Kirkham, M. Luján, and I. Watson. Muts: Native scala constructs for software transactional memory. In *Proceedings of Scala Days*, 2011.

[40] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Luján, and I. Watson. DFScala: High Level Dataflow Support for Scala. In *Data-Flow Execution Models for Extreme Scale Computing*. IEEE, 2012.

[41] J. Gosling. Java Intermediate Bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, 1995.

[42] M. Grimmer. A Runtime Environment for the Truffle/C VM. Master's thesis, 2013.

[43] M. Grimmer. High-performance language interoperability in multi-language runtimes. In *Proceedings of the 2014 Companion Publication for Conference on Systems, Programming, Applications: Software for Humanity*, SPLASH '14, New York, NY, USA, 2014. ACM.

[44] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: dynamic execution of C on a Java virtual machine. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools*, 2014.

[45] M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An Efficient Native Function Interface for Java. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools*, 2013.

[46] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe Execution of C on a Java VM. In *Workshop on Programming Languages and Analysis for Security*, 2015.

[47] M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. Dynamically composing languages in a modular way: supporting C extensions for dynamic languages. In *MODULARITY 2015: Proceedings of the 14th International Conference on Modularity*, pages 1–13, Mar. 2015.

[48] M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An efficient approach for accessing C data structures from JavaScript. In *ICOOOLPS '14: Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2014.

[49] D. H. Hansson et al. Ruby on Rails. http://rubyonrails.org.

[50] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, second edition, 2010.

[51] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005.

[52] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 501–524. Springer Berlin Heidelberg, 2007.

[53] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A path to parallelism in JavaScript. In *ACM SIGPLAN Notices*, volume 48, pages 729–744, 2013.

[54] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer Berlin Heidelberg, 1991.

[55] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, 1992.

[56] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, 2014.

[57] Information-Technology Promotion Agency, Japan. Programming Languages — Ruby, Final Draft. Technical report, 2010.

[58] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. In *Proc. of the Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 179–194, New York, NY, USA, 2012. ACM.

[59] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2013.

[60] Y. Katz. Encodings, Unabridged. http://yehudakatz.com/2010/05/17/encodings-unabridged/, 2010.

[61] B. Keepers. Ruby at GitHub. 2013.

[62] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. In *Programmability Issues for Multi-Core Computer*, 2010.

[63] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of 1st Conference on Virtual Execution Environments (VEE)*, 2005.

[64] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1), 2008.

[65] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[66] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, 1961.

[67] R. LeFevre. OilyPNG. https://github.com/wvanbergen/oily_png.

[68] R. LeFevre et al. PSDNative. https://github.com/layervault/psd_native.

[69] R. LeFevre, K. Sutton, et al. PSD.rb. https://github.com/layervault/psd.rb.

[70] Y. Lin, K. Wang, S. M. Blackburn, A. L. Hosking, and M. Norrish. Stop and Go: Understanding Yieldpoint Behavior. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, pages 70–80, New York, NY, USA, 2015. ACM.

[71] M. Madsen, P. Sørensen, and K. Kristensen. Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm. Master's thesis, Aalborg University, 2007.

[72] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *International Conference on Functional Programming*, 2009.

[73] S. Marr, C. Seaton, and S. Ducasse. Zero-Overhead Metaprogramming. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, 2015.

[74] Y. Matsumoto. Lisp to Ruby to Rubinius. 2010.

[75] B. Mizerany, K. Haase, et al. Sinatra. http://www.sinatrarb.com.

[76] F. Niephaus, M. Springer, T. Felgentreff, T. Pape, and R. Hirschfeld. Call-target-specific Method Arguments. In *Proceedings of the 10th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS)*, 2015.

[77] C. Nutter. So You Want To Optimize Ruby? http://blog.headius.com/2012/10/so-you-want-to-optimize-ruby.html, 2012.

[78] C. Nutter, T. Enebo, et al. JRuby. http://jruby.org/.

[79] R. Odaira, J. G. Castanos, and H. Tomari. Eliminating global interpreter locks in Ruby through hardware transactional memory. In *PPoPP '15: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014.

[80] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, second edition, 2010.

[81] Oracle. Class SwitchPoint, 2015. http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SwitchPoint.html.

[82] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposum*, 2001.

[83] M. Pall et al. LuaJIT. 2015.

[84] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Symposium on Parallel Algorithms and Architectures*, 2011.

[85] P. Perrotta. *Metaprogramming Ruby 2*. Pragmatic Bookshelf, 2014.

[86] E. Phoenix, B. Shirai, et al. Rubinius. http://rubini.us/.

[87] J. Rose, D. Coward, O. Bini, W. Cook, S. Pedroni, and J. Theodorou. JSR 292: Supporting dynamically typed languages on the Java platform, 2008.

[88] J. Rose et al. JSR 292: Supporting Dynamically Typed Languages on the Java Platform, 2011. https://jcp.org/en/jsr/detail?id=292.

[89] J. R. Rose. Bytecodes meet Combinators: invokedynamic on the JVM. In *Proc. of the Workshop on Virtual Machines and Intermediate Languages*, pages 1–11. ACM, Oct. 2009.

[90] K. Sasada. YARV: yet another RubyVM: innovating the ruby interpreter. *OOPSLA Companion*, pages 158–159, 2005.

[91] G. Savrun-Yeniceri, M. L. Van de Vanter, P. Larsen, S. Brunthaler, and M. Franz. An efficient and generic event-based profiler framework for dynamic languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, 2015.

[92] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In *Proceedings of the 8th Workshop on Dynamic Languages and Applications (DYLA)*, 2014.

[93] A. Shali and W. R. Cook. Hybrid Partial Evaluation. In *Proc. of the Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 375–390. ACM, 2011.

[94] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. *VEE*, pages 153–163, 2005.

[95] B. Shirai et al. RubySpec. http://rubyspec.org.

[96] L. Stadler. *Partial Escape Analysis and Scalar Replacement for Java*. PhD thesis, 2014.

[97] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 49–58, New York, NY, USA, 2012. ACM.

[98] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala*, 2013.

[99] L. Stadler, G. Duboscq, T. Würthinger, and H. Mössenböck. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, 2012.

[100] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose. Lazy continuations for Java virtual machines. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, page 143, New York, New York, USA, Aug. 2009. ACM Request Permissions.

[101] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of the Symposium on Code Generation and Optimization (CGO)*, 2014.

[102] M. Stoodley. Multi-language runtime. In *Proceedings of the JVM Language Summit*, 2015.

[103] G. Sullivan. Dynamic Partial Evaluation. In *Programs as Data Objects*, volume 2053 of *LNCS*, pages 238–256. Springer, 2001.

[104] G. Tene, B. Iyengar, and M. Wolf. C4: the continuously concurrent compacting collector. In *ISMM '11: Proceedings of the international symposium on Memory management*, pages 79–88, New York, New York, USA, June 2011. ACM Request Permissions.

[105] C. Thalinger. Java goes aot. In *Proceedings of the JVM Language Summit*, 2015.

[106] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[107] W. van Bergen et al. Chunky PNG. https://github.com/wvanbergen/chunky_png.

[108] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? In *ACM Asia-Pacific Workshop on Systems*, 2012.

[109] I. Watson, C. Kirkham, and M. Luján. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[110] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: a parallel architecture for declarative programming. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, 1988.

[111] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *Transactions on Architecture and Code Optimization (TACO*, 9(4):1–24, Jan. 2013.

[112] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, and C. Humer. An object storage model for the Truffle language implementation framework. In *PPPJ '14: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 133–144, Sept. 2014.

[113] T. Würthinger. *Visualization of Program Dependence Graphs*. PhD thesis, Aug. 2007.

[114] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Onward! '13: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.

[115] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic languages*, 2013.

[116] M. Yukihiro et al. Matz's Ruby Interpreter. https://www.ruby-lang.org/.