Efficient Interpreters and Profilers for Hosted Dynamic Languages

Permalink
https://escholarship.org/uc/item/3025h4p9

Author
Savrun Yeniceri, Gulfem

Publication Date
2015-01-01

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Interpreters and Profilers for Hosted Dynamic Languages

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Gülfem Savrun Yeniçeri

Dissertation Committee:
Professor Michael Franz, Chair
Professor Tony Givargis
Professor Guoqing Xu

2015
DEDICATION

I dedicate this dissertation to my dearest husband Onur, and my lovely son Bora. I thank my husband for all the love, support, and encouragement he has given during the challenges of graduate school and life. I love you more than anything. I thank my baby Bora for all the joy he brought to my life. I love you my little Bora, and I will always be there to hold your hand as I did from the first moment I saw you. I also dedicate my dissertation to my wonderful parents, Rukiye and Zafer Savrun. Thank you very much for raising me, teaching me all the good things in life, and loving me unconditionally.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF ALGORITHMS</td>
<td>viii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>x</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>xiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Interpreters</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Switch-based Dispatch</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Direct Call Threaded Code</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3 Direct Threaded Code</td>
<td>8</td>
</tr>
<tr>
<td>2.1.4 Indirect Threaded Code</td>
<td>9</td>
</tr>
<tr>
<td>2.1.5 Subroutine Threaded Code</td>
<td>9</td>
</tr>
<tr>
<td>2.1.6 Context Threaded Code</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Compilers</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1 Ahead-Of-Time Compilers</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Just-In-Time Compilers</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Profilers</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1 Event-based Profilers</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2 Sampling-based Profilers</td>
<td>14</td>
</tr>
<tr>
<td>3 Efficient Hosted Interpreters</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Efficient Instruction Dispatch</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Efficient Array Stores</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Modular VM</td>
<td>22</td>
</tr>
<tr>
<td>3.3.1 The Hosted Language Implementer’s Perspective</td>
<td>24</td>
</tr>
<tr>
<td>3.3.2 The Java Virtual Machine Implementer’s Perspective</td>
<td>24</td>
</tr>
</tbody>
</table>
# Efficient Profiler Framework

4.1 Truffle Framework ................................................................. 35
4.2 Instrumentation Framework ..................................................... 37
4.3 Event-based Profiling ............................................................ 38
4.4 Method Profiling ................................................................. 41
4.5 Cross-language Comparison ................................................. 42

5 Evaluation

5.1 Efficient Hosted Interpreters ................................................ 44
5.1.1 Experimental Setup ........................................................ 45
5.1.2 Comparison with Switch-based Interpreter ............................ 46
5.1.3 Comparison with Host VM Targeted Compiler ......................... 49
5.1.4 Comparison with HotSpot Server Compiler ............................ 52
5.1.5 Implementation Effort Comparison ........................................ 54
5.1.6 Array Store Check Frequency ............................................. 55
5.1.7 Discussion ................................................................. 57
5.2 Efficient Profiler Framework ................................................ 59
5.2.1 Experimental Setup ........................................................ 59
5.2.2 Event-based Profiling Performance ....................................... 60
5.2.3 Method Profiling Performance ............................................ 62
5.2.4 Cross-language Comparison .............................................. 65
5.2.5 Profiler Implementation Effort Comparison ............................ 66

6 Related Work

6.1 Threaded Code ................................................................. 68
6.2 Targeting Virtual Machines .................................................. 71
6.3 Profilers ............................................................... 72

7 Conclusion

Bibliography

A Efficient Hosted Interpreters Detailed Benchmark Results 86
A.1 Jython Results ................................................................. 87
A.2 Rhino Results ................................................................. 90
A.3 JRuby Results ................................................................. 93

B Efficient Profiler Framework Detailed Benchmark Results 96
B.1 Python Results ................................................................. 97
B.2 Ruby Results ................................................................. 98
# LIST OF FIGURES

1.1 Evolution of Dynamic Languages and Their Implementations.  
2.1 Switch-based Dispatch.  
2.2 Direct Call Threaded Code.  
2.3 Direct Threaded Code.  
2.4 Indirect Threaded Code.  
2.5 Subroutine Threaded Code.  
2.6 Context Threaded Code.  
3.1 Hosted VM approach.  
3.2 Jython’s hosted interpreter vs. host VM targeted compiler.  
3.3 Execution components of a BINARY_ADD instruction in Jython.  
3.4 Operand stack in Jython.  
3.5 Object model of Jython.  
3.6 Annotation-based system.  
3.7 Extracting instruction operations into methods and adding annotations in Jython.  
3.8 Initialization phase of threaded code generator in Jython.  
3.9 Threaded code generation phase in Jython.  
3.10 Operand argument in Python.  
3.11 Handling operand arguments in Jython.  
3.12 Handling unconditional branch in Jython.  
3.13 Handling conditional branch in Jython.  
3.14 Handling return in Jython.  
3.15 An example in Jython.  
4.1 AST of an add function before and after inserting instrumentation nodes to the add operation event.  
4.2 Typed versions of an add node in Truffle.  
4.3 Implementation of instrumentation nodes.  
4.4 Implementation of an event profiler instrument node.  
4.5 Implementation of a method profiler instrument node.  
5.1 Jython.  
5.2 Rhino.  

Page

2
6
7
8
9
10
11
16
17
19
21
21
23
25
26
27
28
28
29
30
30
32
36
37
39
40
42
47
47
5.3 JRuby. ................................................................. 47
5.4 Performance of subroutine threaded code interpreter relative to switch-based
interpreters. .......................................................... 47
5.5 Jython. ............................................................... 50
5.6 Rhino. ............................................................... 50
5.7 JRuby. ............................................................... 50
5.8 Performance of subroutine threaded code interpreter relative to host VM tar-
geted compilers. ...................................................... 50
5.9 Jython. ............................................................... 53
5.10 Rhino. ............................................................... 53
5.11 JRuby. ............................................................... 53
5.12 Performance of compiling switch-based interpreters with Java Hotspot’s server
compiler relative to client compiler. ................................. 53
5.13 Event-based profiling in ZipPy. .................................. 61
5.14 Event-based profiling in JRuby+Truffle. ........................ 61
5.15 Method profiling comparisons for Python. ...................... 64
5.16 Method profiling comparisons for Ruby. ........................ 64
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Performance of subroutine threaded code interpreter relative to switch-based interpreters.</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance of subroutine threaded code interpreter relative to host VM targeted compilers.</td>
</tr>
<tr>
<td>5.3</td>
<td>Hosted interpreter vs. host VM targeted compiler implementation effort.</td>
</tr>
<tr>
<td>5.4</td>
<td>Array store check frequencies in millions for the DaCapo benchmarks.</td>
</tr>
<tr>
<td>5.5</td>
<td>Array store check frequency in millions for Jython’s subroutine threaded code interpreter.</td>
</tr>
<tr>
<td>5.6</td>
<td>Total number of executed events in ZipPy.</td>
</tr>
<tr>
<td>5.7</td>
<td>Total number of executed events in JRuby+Truffle.</td>
</tr>
<tr>
<td>5.8</td>
<td>Profiler implementation effort comparison.</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Initialization Phase ........................................ 26
2. Threaded Code Generation Phase ......................... 27
3. Execution of Threaded Code ............................... 31
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Dr. Michael Franz, for his supervision. I am deeply grateful that he admitted me to his amazing research group and brought me into the world of programming languages, interpreters, and compilers.

I would like to thank Professor Tony Givargis and Guoqing Xu who accepted to serve on my committee. I am very grateful to my Postdoctoral fellows, Stefan Brunthaler and Per Larsen, for all the feedback, guidance and motivation. This dissertation would not have been possible without their help and support. I would like to express my gratitude to Michael Bebenita and Mason Chang for introducing me into virtual machines, interpreters, and compilers. I have learned more than I expected from them. Their enthusiasm and encouragement made me to get interested in virtual machines. I would like to thank my teammate Wei Zhang, for all the work that we did together. Furthermore, I would like to thank to my wonderful labmates which made graduate school a truly wonderful experience, especially: Gregor Wagner, Todd Jackson, Eric Hennigan, Christoph Kerschbaumer, Andrei Homescu, Stephen Crane, Codrut Stancu, Mark Murphy, Mohaned Qunaibit, Brian Belleville, and Julian Lettner. Lastly, I want to thank Michael L. Van de Vanter for his support and insightful comments.

Parts of this dissertation were previously published at the following venues:


This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N66001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Oracle Labs. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.
CURRICULUM VITAE

Gülfem Savrun Yeniçeri

EDUCATION
Doctor of Philosophy in Computer Science  2015
University of California, Irvine  GPA: 3.99  Advisor: Michael Franz  Irvine, CA

Master of Science in Computer Science  2010
New York University  GPA: 3.61  Advisor: Benjamin Goldberg  New York, NY

Bachelor of Science in Computer Engineering  2007
Minor in Telecommunications
Mersin University  Rank 1 in Faculty of Engineering  Mersin, TURKEY

RESEARCH INTERESTS
Programming languages, virtual machines, compilers, interpreters, runtime systems, optimizations, software engineering

PUBLICATIONS
Journal Publications
doi: http://dx.doi.org/10.1145/2532642.

Conference Publications
doi: http://dx.doi.org/10.1145/2807426.2807435.

doi: http://dx.doi.org/10.1145/2500828.2500839.
PRESENTATIONS
P2. Efficient Interpreter Optimizations for the JVM 2013
International Conference on Principles and Practice of Programming in Java (PPPJ)

P1. It’s About Time: JITing Threaded Code 2012
The SoCal Programming Languages and Systems Workshop

TEACHING EXPERIENCE
University of California, Irvine
CS142B, Compiler Construction Project, Teaching Assistant Winter 2013
CS143A, Principles of Operating Systems, Teaching Assistant Winter 2013
CS142A, Compilers and Interpreters, Teaching Assistant Spring 2013

WORK EXPERIENCE
Implemented a generic, high-performance profiler for Python and Ruby implementations using Oracle’s Truffle framework and the Graal Just-In-Time (JIT) compiler

Implemented Python language on the Java Virtual Machine (JVM) using Oracle’s Truffle framework and the Graal Just-In-Time (JIT) compiler

Modular VM: Fine-Grained Modularity & Reuse of VM Components 2011 - 2013
Implemented an efficient hosted interpreter mechanism for Jython (Python), JRuby (Ruby) and Rhino (JavaScript) dynamic language implementations running on a Java Virtual Machine (JVM)

PROJECTS
Lexer, Parser, SSA intermediate representation generator, SSA-based optimizations, register allocator, and x86 machine code generator for an advanced compiler
Record manager, indexing, and query processor modules of a database management system
Web development project with Ruby on Rails
Scheme mini-interpreter
Polymorphic type inferencer in ML
Syntax-driven translator from English to Turkish in Prolog
Simple, yet scalable web server
Distributed replicated concurrency control and recovery system
Open source tools project using Perl, CGI scripting, XML and MySQL
Toy OS that schedules processes and uses paging for memory isolation
RSA public/private crypto system for creating digital certificates and authentication
SELECTED HONORS AND AWARDS
Society of Women Engineers Graduate Student Scholarship 2015
Joseph and Dorothy Fischer Endowed Memorial Fellowship Award 2014
Building Recruiting And Inclusion for Diversity (BRAID) Scholarship 2014
Grace Hopper Celebration of Women in Computing (GHC) Scholarship Grant 2013
NSF-sponsored Travel Grant for ASPLOS Conference 2011
Turkish Government Scholarship - Graduate Education in the USA 2008
Istanbul Metropolitan Municipality Graduate Education Scholarship 2007
Top Undergraduate Student of Faculty of Engineering from Mersin University 2007
4 High Honor Certificates and 4 Honor Certificates from Mersin University

ACTIVITIES
Organizer of Graduate Talk Series, University of California, Irvine
Member of ACM, ACM-W, Society of Women Engineers
Member of Women in Information & Computer Sciences, University of California, Irvine
Member of Women in Computing (WINC), New York University
Member of Turkish Women in Computing (TWIC)
ABSTRACT OF THE DISSERTATION

Efficient Interpreters and Profilers for Hosted Dynamic Languages

By

Gülfem Savrun Yeniçeri

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Michael Franz, Chair

Dynamic languages such as Perl, Python, JavaScript, Ruby, and PHP are becoming increasingly popular. Many dynamic language implementers choose a layered approach, where a guest language runs on a host virtual machine (VM), while implementing their language. Implementers have two possible execution choices on a host VM: hosted interpreters and host VM targeted compilers. Hosted interpreters run as a regular application on a host VM, whereas, host VM targeted compilers translate the hosted language programs to host VM’s instruction set, and execute it on the host VM. Hosted interpreters are comparatively simpler to implement but suffer from poor performance. On the other hand, host VM targeted compilers are complex to build but lead to better performance.

In this dissertation, we explore hosted dynamic language interpreters targeting the Java virtual machine (JVM). The purpose of our research is to close the performance gap between hosted interpreters and host VM targeted compilers. Next, we implement a generic, high-performance profiler framework for hosted dynamic languages on a Java virtual machine with modest implementation effort. Our framework has a unique feature, which makes it possible to compare and analyze the programs implementing the same algorithms across different dynamic languages.
Chapter 1

Introduction

Dynamic languages such as Python, Ruby, JavaScript, and PHP started to appear around 1990s as shown in Figure 1.1. They are sometimes referred as *scripting languages* because they were originally designed to write small scripts to automate the execution of tasks while building a large system. With the development and growth of World Wide Web in 1990s, dynamic languages have become pervasive in web programming. PHP, Python and Ruby are used in server-side scripting, while PHP being the dominant language. Whereas, JavaScript is the most popular language on client-side.

Programmers now use dynamic languages to build large scale applications. For example, YouTube [16], Dropbox [1], Yelp [15], Quora [10] and Reddit [11] use Python, and Twitter [14], GitHub [3] and Hulu [4] use Ruby in their implementation. Similarly, Django and Ruby on Rails are popular high-level web application frameworks in Python and Ruby, respectively.

Dynamic languages have gained popularity to the point that their performance is starting to matter. Implementing a production-quality, high-performance runtime for a specific dynamic language requires significant development effort and investment. However, this is often not
available to many dynamic language communities. Therefore, many language implementers choose a hosted virtual machine approach by implementing their language on a host virtual machine. If a dynamic language implementation runs on top of a widely available virtual machine such as the Java virtual machine, the new language then becomes instantly available on all the platforms that already have the underlying virtual machine.

Virtual machines are virtualized programming environments for high-level programming languages. They provide a platform-independent environment by abstracting away the specifics of the architecture and operating system running beneath them. Virtual machines are also called Managed Runtime Environments (MRE) because they handle runtime services, such as automatic memory management, concurrency management, compilation, and exception handling. Because of the runtime’s close relationship with the architecture and operating system, they are referred as a virtual machine. Virtual machines have become popular with the development of the Java virtual machine. They are now widely used to implement
high-level programming languages, such as C#, Scala, and Python.

Improving the performance of Java virtual machines has been an active research area since 2000s. Oracle’s Java HotSpot virtual machine [13] and IBM’s J9 virtual machine [55, 50] are production Java virtual machines using many complex optimization techniques. Java virtual machines are sophisticated and mature pieces of software which are designed to improve performance and security. Because of portability, performance, and security benefits, the Java virtual machine is a popular host environment for dynamic languages. For example, Jython [7], JRuby [5], and Rhino [12] are Python, Ruby, and JavaScript implementations on top a Java virtual machine, respectively.

The easiest way to get a new dynamic language off the ground on a Java virtual machine is by writing a hosted interpreter in Java. Due to the serious performance implications of hosted interpreters, implementers write host VM targeted compilers that emit Java byte-code instead. This dissertation investigates the performance potential of optimizing hosted interpreters on the Java virtual machine, aiming to combine the simplicity of the hosted interpreters with the performance of host VM targeted compilers. We studied the performance of hosted interpreters, identified common bottlenecks preventing their efficient execution, and presented optimizations to improve their performance. In chapter 3, we explain the details of our system that improves the performance of hosted interpreters.

As the number and popularity of dynamic language applications grow, optimizing their performance and building efficient analysis tools such as profilers for them become important. Profilers are important program analysis tools that help programmers analyze their programs and identify performance bottlenecks. This dissertation focuses on building a generic profiler framework for hosted dynamic languages on a Java virtual machine to minimize the recurring implementation effort. Our profiler framework makes it possible to compare and evaluate different dynamic language programs executing the same algorithms. In chapter 4, we explain the details of our profiler framework.
Chapter 2

Background

Virtual machines have two typical execution mechanisms: interpreters and compilers. Interpreters interpret the program by executing one statement at a time. On the other hand, compilers compile the program’s source code into native machine code, and execute it on the target architecture. A third option is to have a mixed mode execution strategy which uses an interpreter as the first level of execution and a compiler as the second level of execution. An interpreter starts executing the program, and a compiler compiles performance critical regions to improve performance at runtime. Many industrial-strength virtual machines, such as the Java HotSpot virtual machine [13], use this hybrid execution strategy. This chapter explains the characteristics of interpreters and compilers in a virtual machine.

2.1 Interpreters

Interpreters have many benefits which make them appealing to the language implementers:

- Interpreters are easier to implement and maintain than compilers.
• Interpreters are more flexible than compilers because it is easier to modify, examine, and collect diagnostic information about the structure and behavior of a program in an interpreter.

• Interpreters are more portable than compilers, so porting an interpreter to another platform requires little implementation effort.

• Interpreters use less memory than compilers.

• Interpreters have a shorter startup time than compilers. They start execution immediately, whereas, compilers first compile a program, causing higher latency.

Virtual machine interpreters can be divided into two main types based on the intermediate format that they interpret:

• Abstract syntax tree (AST) interpreters

• Bytecode interpreters

In an AST interpreter, a parser parses the program using the grammar of the language and builds an AST. Then, an AST interpreter interprets the resulting AST by adding execution semantics to each AST node. Each node execution performs an expensive tree traversal which incurs significant performance overhead. For faster execution, language implementers translate source programs into defined bytecode, and write a bytecode interpreter, operating on the bytecode.

Bytecode interpreters need to fetch, decode and execute instructions like a processor. They perform a process called instruction dispatch for every instruction, which includes fetching the instruction from the instruction stream, decoding the instruction opcode, and transferring the control to the instruction implementation. Instruction dispatch is on the critical path and its performance greatly affects overall interpreter performance. As Ertl and Gregg
pointed out, it is the most important bottleneck for bytecode interpreters [46]. Therefore, researchers have proposed several techniques to optimize instruction dispatch.

2.1.1 Switch-based Dispatch

Switch-based dispatch is the simplest dispatch technique. It nests a `switch` statement inside a loop, and each `case` statement implements an instruction as shown in Figure 2.1. Typically, the compiler will generate an address table for all the `case` blocks and use an indirect branch instruction to find the address for the current opcode when it compiles a `switch` statement. Therefore, all interpreter instructions share just one indirect branch. Since the target for this indirect branch depends entirely on the instruction stream of the interpreted program and not on the machine context, the branch prediction will fail most of the time. The high probability of branch misprediction limits the performance of switch-based dispatch.

![Figure 2.1: Switch-based Dispatch.](image-url)
2.1.2 Direct Call Threaded Code

Direct call threaded code [43] optimizes instruction dispatch by eliminating the switch statement. It implements each instruction in a separate function and uses a function pointer to dispatch instructions inside a loop as shown in Figure 2.2. Switch-based dispatch operates on the opcodes of the instructions, whereas, direct call threaded code operates on the addresses of the functions that implements the instructions. Although the technique is called direct call threaded code, it uses indirect calls for instruction dispatch. Historically, the name indirect is given to interpreters that use a mapping to convert an instruction opcode to the address of the instruction implementation like in switch-based dispatch. On the other hand, the name direct is used for interpreters operating directly on the addresses of the instruction implementations.

Figure 2.2: Direct Call Threaded Code.
2.1.3 Direct Threaded Code

Direct threaded code [21] is a technique that eliminates the dispatch loop used in switch-based dispatch and direct call threaded code. It converts the instruction stream into the actual addresses of the instruction implementations. Interpretation starts by fetching the first address and then jumping to that address. Each instruction implementation performs its work, and ends with a jump, transferring control to the next instruction. Figure 2.3 shows the implementation of direct threaded code using the computed goto extension in GNU C. The extension allows to obtain the address of a label in a function using the unary operator &&, and jump to that address using the goto statement. Direct threaded code spreads out the instruction dispatch to the end of each instruction implementation that improves branch prediction. Instead of using one shared indirect branch as in switch-based dispatch, there are multiple indirect branches, giving branch prediction more context in direct threaded code. If, for example, instruction B is likely to follow instruction A, then the likelihood of correct branch prediction increases.

![Diagram of Direct Threaded Code]

Figure 2.3: Direct Threaded Code.
2.1.4 Indirect Threaded Code

Indirect threaded code [41] reduces memory used in direct threaded code by using one level of indirection. It maps instruction opcodes to the instruction implementation addresses. Indirect threaded code dispatches next instruction by reading the instruction opcode, finding its implementation address, and jumping to the target address as shown in Figure 2.4. Loaded programs consist of memory addresses in direct threaded code. Whereas, they consist of instruction opcodes, which are typically encoded in one byte in indirect threaded code. This makes indirect threaded code more compact than direct threaded code, but the indirection makes it slower.

```
interpret() {
      add:
          …
          goto *ITCT[*pc++];
      load:
           ...
          … load
...
 store
...
 load
 add
Indirect Threaded Code Table (ITCT)
 &&load
...
 &&store
...
 &&add
```

![Figure 2.4: Indirect Threaded Code.](image)

2.1.5 Subroutine Threaded Code

Subroutine threaded code optimizes instruction dispatch by emitting executable code. It implements each instruction in a separate subroutine, and translates the instruction stream into machine-level calls to these subroutines as shown in Figure 2.5. Direct threaded code
uses jumps for instruction dispatch, whereas, subroutine threaded code uses calls. Since the whole instruction stream consists of a series of native machine call instructions, it is sometimes referred as call threaded code. Each instruction execution performs a call to a subroutine, and transfers the control back to the instruction stream via a machine return instruction. This return instruction is highly predictable, as modern CPUs use a return address stack for exactly this purpose.

![Figure 2.5: Subroutine Threaded Code.](image)

### 2.1.6 Context Threaded Code

Context threaded code [23] is an improvement over subroutine threaded code. Figure 2.6 shows the implementation of context threaded code. Context threaded code introduces branch replication and branch inlining to improve branch prediction further and reduce pipeline branch hazards. In subroutine threaded code, a branch instruction implementation translates to one shared native indirect branch instruction. This causes poor branch prediction for the control-flow changing instructions in the interpreter. Branch replication adds a native indirect branch instruction into subroutine threaded code, causing each branch in-
struction has its own branch predictor entry. Context threaded code uses branch replication for indirect branches and exceptions. It uses branch inlining for other types of branches which inlines the branch instruction implementation into subroutine threaded code. This eliminates the overhead of calling the associated branch instruction implementation. Moreover, context threaded code implements Tiny Inlining, which uses a simple heuristic to inline small instruction implementations into threaded code to further reduce dispatch overhead.

2.2 Compilers

Compilers translate source code into destination code. Destination code can be virtual machine code or native machine code. There are typically two separate compilers for the languages running on a virtual machine. One compiler translates source code into virtual machine code, and another compiler turns virtual machine code to native machine code. For example, the javac compiler translates Java source code to Java bytecode before execution.
An architecture specific compiler in a Java virtual machine translates Java bytecode to native machine code at runtime. Similarly, Microsoft’s .NET Framework compiles source programs to Common Intermediate Language (CIL) code. Upon execution, Common Language Runtime (CLR) [68], the virtual machine component of Microsoft’s .NET framework, turns CIL code into native machine code.

Compilers are typically divided into two categories based on when execution is performed:

- **Ahead-of-time (AOT) compilers**
- **Just-in-time (JIT) compilers**

### 2.2.1 Ahead-Of-Time Compilers

Ahead-of-time compilers translate source programs to machine code prior to execution. For instance, the javac compiler performs ahead-of-time, transforming Java source programs to Java bytecode before runtime. Ahead-of-time compilers perform complex and time-consuming compiler optimizations to produce optimized machine code.

### 2.2.2 Just-In-Time Compilers

Just-in-time compilers translate source code during execution. For example, the Java virtual machine has a just-in-time compiler which compiles Java bytecode into native machine code at runtime. Just-in-time compilers are time-constrained, so they have to make a trade-off between the compilation time and the optimizations that they can perform. The main advantage of the just-in-time compilers is that they have much more information available than an ahead-of-time compilers. Just-in-time compilers use a technique called adaptive optimization [53, 17] which collects runtime information and use it for optimizations decisions.

2.3 Profilers

Program analysis tools are important for understanding program behavior. Profilers are a type of dynamic program analysis tool that measure a program’s runtime characteristics and resource utilization. For example, it measures memory or time complexity of a program to identify which areas of a program offer the greatest potential performance increase. It analyzes the program’s performance for *hot spots* which are frequently or repeatedly executed, and finds bottlenecks that uses a program’s resources inefficiently. The goal is to reduce resource usage, and tune the performance.

It is common to classify profiling tools into two major categories:

- Memory profilers
- Performance profilers

Memory profilers are used to solve memory related issues like memory leaks, and high memory consumption. Performance profilers are used to solve performance related issues like high CPU usage or concurrency related problems. Performance profilers are further classified into another two categories:

- Event-based profilers
- Sampling-based profilers (also called statistical profilers)
2.3.1 Event-based Profilers

Event-based profilers track every occurrence of certain events from a well-defined event set. Events may include method invocation, object allocation, exception and so on. Event-based profilers typically use instrumentation to modify the source code or binary code of a program. They collect data every step of the way while the application is running. Event-based profilers produce more accurate results, but additional inserted probes significantly affect the performance of the running application.

2.3.2 Sampling-based Profilers

Sampling-based profilers interrupt the CPU at regular intervals to collect data samples about the work performed by an application. The resulting data of sampling-based profilers is not exact, but a statistical approximation. Sampling is typically lightweight and has little effect on the execution of the running programs.
Chapter 3

Efficient Hosted Interpreters

Virtual machines are typically developed entirely from scratch for a specific language. For example, the Java virtual machine was originally designed to support the Java programming language. With the popularity of the Java platform, many languages were adapted or designed to run on the Java virtual machine.

The rise of dynamic languages has led to a diversity of virtual machines. There are some design approaches to implement a virtual machine for dynamic languages. One approach is to develop a completely new virtual machine for a specific dynamic language. This approach typically contributes to good performance because the virtual machine is carefully designed for optimizations and specializations targeting that language. For example, V8 JavaScript engine [48] is designed for the fast execution of JavaScript applications in the Google Chrome web browser. It has a native compiler that translates JavaScript source code into executable machine code. The problem with this approach is that it requires significant development and maintenance effort, and is not scalable due to the variety of dynamic languages.

Another approach to dynamic language virtual machine implementation is to develop a virtual machine on top of a widely available host virtual machine. Figure 3.1 shows an
example of this *hosted* approach. In this example, Java virtual machine is the underlying *host virtual machine*, and Jython is the *hosted virtual machine*. Jython executes Python programs on top of a Java virtual machine.

In this hosted approach, implementers have two major execution choices. First choice is called *hosted interpreter*, which is written in the host language that the underlying host VM supports. For our example, host language is Java, and Jython’s hosted interpreter is implemented in Java. Second choice is *host VM targeted compiler* that translates source programs to the underlying host VM’s instruction set. In our example, host VM’s instruction set is Java bytecode, so Jython’s host VM targeted compiler translates Python programs to Java bytecode, and runs it on a Java virtual machine.

Jython’s hosted interpreter is switch-based interpreter, and it is a straightforward port of an interpreter present in CPython [77], the reference implementation of Python in C. Similar to CPython interpreter, Jython’s hosted interpreter operates on the Python bytecode. Jython’s parser parses Python programs, and generates AST nodes. The hosted interpreter performs another step in the front end by translating AST nodes to Python bytecode before execution. At runtime, the interpreter executes one Python bytecode at a time. On the other hand, Jython’s host VM targeted compiler translates AST nodes to Java bytecode at runtime, and subsequently executes it on a Java virtual machine. Figure 3.2 shows the details of Jython’s hosted interpreter and Jython’s host VM targeted compiler.
We study the hosted bytecode interpreters running on a Java virtual machine in this chapter. We introduce our Modular VM\(^1\) system that significantly improves the performance of these interpreters. Hosted bytecode interpreters have not yet delivered good performance, therefore, many implementers implement a host VM targeted compiler which compiles source language to Java bytecode. We investigate the reasons limiting the performance of hosted bytecode interpreters executing on a Java virtual machine. We identify two common bottlenecks preventing their efficient execution:

1. Instruction dispatch
2. Array store checks

We present our optimizations targeting these two bottlenecks, and evaluate their performance on three hosted dynamic language virtual machines: Jython, Rhino and JRuby, Python, JavaScript, and Ruby implementations respectively.

---
\(^1\)Modular VM is available at [https://bitbucket.org/thezhangwei/modularvm](https://bitbucket.org/thezhangwei/modularvm)
3.1 Efficient Instruction Dispatch

Bytecode interpreters perform the following two steps while executing each instruction:

- Instruction dispatch
- Instruction operation

Instruction dispatch constitutes the interpreter overhead, whereas, instruction operation constitutes the actual computation of the program. Instruction dispatch controls the execution transfer from one instruction to the next instruction. Instruction operation performs the functionality in the instruction implementation. Reducing the cost of instruction dispatch increases the time spent in doing the actual work of the program. Instruction dispatch consists of three components:

- Instruction fetch
- Instruction decode
- Control transfer to the instruction operation

Figure 3.3 displays the components of an execution of a \texttt{BINARY\_ADD} instruction, which adds two values, in Jython. \texttt{BINARY\_ADD} instruction pops two values from the stack, performs the addition, and pushes the result back to the stack.

Bytecode interpreters using a switch-based dispatch suffer from poor performance due to the reasons explained in subsection 2.1.1. Threaded code variations are the efficient dispatch techniques that reduce the cost of instruction dispatch as described in chapter 2. In C, \textit{labels-as-values} and \textit{computed goto} extensions make it possible to implement threaded code. However, all of the machinery required to create and execute threaded code is not available
while (true) {
    int opcode = bytecode[next_instr++];
    ...
    switch (opcode) {
        ...
        case Opcode.BINARY_ADD:
            PyObject b = stack.pop();
            PyObject a = stack.pop();
            stack.push(a._add(b));
            break;
        ...
    }
    ...
}

Figure 3.3: Execution components of a BINARY_ADD instruction in Jython.

to Java programmers. There is no way to program indirect branches in Java, such as those generated by computed goto instructions in C. As a result, a hosted bytecode interpreter written in Java can not make use of the threaded code techniques. We address this issue by adding the ability to generate threaded code to the underlying Java virtual machine in our Modular VM.

3.2 Efficient Array Stores

There are two main kinds of virtual machine instruction sets:

- Stack-based instruction set
- Register-based instruction set

The Java bytecode and Python bytecode are stack-based instruction sets. The Dalvik virtual machine and Lua virtual machine use register-based instruction sets. Stack-based instruction sets use a data structure called the operand stack, holding the operands used by operators to perform operations. Instructions take operands from the operand stack, operate on them,
and push the result back onto the operand stack. Usually, interpreter implementers use an array to implement this operand stack. Thus, pushing an operand corresponds to writing to the array, and popping an operand to reading from the array.

In a stack-based instruction set, almost every instruction is going to push its result on the array. Shi et al. [85] measured that load instructions for Java bytecode account for almost half of all executed interpreter instructions. Similar measurements for Python confirm that this observation holds for Python bytecode interpreters, too [32]. In consequence, we establish that due to this unusually high frequency, efficient interpreters need high array store performance.

In systems programming languages, such as C and C++, arrays are implicitly low-level and therefore yield the desired performance. But in Java, array semantics are different and it dictates expensive runtime exception checks, which negatively affect array performance essential to interpreters. In Java, storing an element to an object array can possibly throw three exceptions:

- **NullPointerException**
- **ArrayIndexOutOfBoundsException**
- **ArrayStoreException**

First, if the reference to an array is null, it throws a `NullPointerException`. Second, if the index is not within the bounds of the array, it throws an `ArrayIndexOutOfBoundsException`. Third, if the actual type of value is not assignment compatible with the actual type of the components of the array, it throws an `ArrayStoreException`. This type check is more expensive than the other two exception checks because it performs a type comparison by traversing the inheritance hierarchy, which might be very deep.
In hosted bytecode interpreters, almost all instructions write the results of their execution to the operand stack, therefore, repeatedly verifying the type-safety of the array used to model the operand stack is expensive. Figure 3.4 shows the implementation of the operand stack, and two instructions operating on the operand stack in Jython. Jython uses `PyStack` to manage the operand stack. Internally, `PyStack` uses an array of `PyObject` objects to implement a stack. `PyObject` is the root class of Jython’s object hierarchy used to model Jython’s object model on the Java virtual machine. During actual interpretation, elements of the `stack` array will be instances of different Python types, such as `PyInteger`, `PyFloat`, `PyString`, and `PyList`, as shown in Figure 3.5. As a result, type checking the `stack` Java array requires repeatedly verifying that the objects actually derive from `PyObject`.

```java
PyObject interpret(PyFrame f, ...) {
    PyStack stack = new PyStack(stacksize);
    ...
    while (true) {
        int opcode = bytecode[next_instr++];
        ...
        switch (opcode) {
            ...
            case Opcode.LOAD_FAST:
                stack.push(f.getlocal(oparg));
                break;
            case Opcode.BINARY_ADD:
                PyObject b = stack.pop();
                PyObject a = stack.pop();
                stack.push(a._add(b));
                break;
            ...
            ...
        }
    }
    ...
}
```

```

class PyStack {
    PyObject[] stack;
    void push(PyObject v) {
        stack[++top] = v;
    }
    PyObject pop() {
        return stack[top--];
    }
}
```

Figure 3.4: Operand stack in Jython.

```
PyObject PyInteger PyFloat PyString PyList ...
...
```

Figure 3.5: Object model of Jython.
It turns out, however, that checking this exception is completely *redundant*. Since `PyObject` is the root class for all Jython-level classes, it follows that a sound interpreter implementation will exclusively operate on objects corresponding to this class hierarchy. Consequently, while checking the `ArrayStoreException` is necessary in the general case, it is strictly not necessary for a hosted bytecode interpreter operating on its own class hierarchy. Put differently, by construction the interpreter will never operate on an object not deriving from `PyObject`. Note that the same argument holds not only for Jython, but for other implementation, too, such as Rhino and JRuby, which we use in our evaluation.

While it is known that exception checks are expensive, and that exception elimination has been actively researched and successfully addressed in previous work, the case for virtual machine interpreters is particularly pathological. The reason for this is twofold: (i) the unusually high frequency of array stores on the operand stack, and (ii) expensive nature of involved type checking operations. To address this issue, we eliminate redundant `ArrayStoreException` checks in our Modular VM, which boosts the performance of the investigated interpreters.

### 3.3 Modular VM

We present our Modular VM [92, 82, 83] system, which significantly improves the performance of hosted bytecode interpreters running on a Java virtual machine. It optimizes the two bottlenecks mentioned in the previous sections as follows:

1. Optimizes instruction dispatch by generating subroutine threaded code
2. Optimizes arrays by eliminating array store checks

In this section, we are going to discuss our implementation of reusable, annotation-based interpreter optimizations addressing both issues. By adding these optimizations to the Java
virtual machine, they become immediately available to all hosted language implementations. We explain our implementation in two perspectives:

- The hosted language implementer’s perspective
- The Java virtual machine implementer’s perspective

We respect the abstraction layers of hosted virtual machine and host virtual machine, and use optional annotations to communicate between these two layers in our implementation as shown in Figure 3.6. While using our system, a hosted language implementer should write a modified version of switch-based interpreter that potentially works on any Java virtual machine, and provides some information to the underlying Java virtual machine through annotations. The underlying Java virtual machine uses these annotations to generate an efficient interpreter. An annotation-unaware Java virtual machine ignores the annotations, and executes the modified hosted language interpreter as it is. We require the hosted language implementation effort to be as little as possible, and push much of the complexity to the Java virtual machine implementer.

![Figure 3.6: Annotation-based system.](image-url)
3.3.1 The Hosted Language Implementer’s Perspective

Our current full-fledged prototype implementation requires only negligible sets of changes for a hosted language implementer. First, we assume that the language implementer already implemented a switch-based interpreter. Frequently, this resembles a straightforward port of an interpreter present in a system that does not target the Java virtual machine. For example, Jython’s bytecode interpreter resembles the CPython’s interpreter. For the hosted language implementer to enable threaded code generation and execution, we require only three steps:

1. Extract the instruction operations to their own methods
2. Add annotations to these methods
3. Replace the case bodies with method calls to the extracted methods

Figure 3.7 shows this transformation for BINARY_ADD and BINARY_SUBTRACT instructions in Jython. It is worth noting that this transformation is purely mechanical. This is precisely, why we believe that this transformation could be automated in future work. For example, by annotating only the dispatch loop, one should be able to automate the subsequent processing steps. But, even without automation, this task is straightforward.

3.3.2 The Java Virtual Machine Implementer’s Perspective

The Java virtual machine implementer needs to add our threaded code generator to her implementation. Our threaded code generator consists of two phases:

1. Initialization phase
2. Threaded code generation phase
Initialization Phase

In the initialization phase, we find all the instruction operation methods extracted and annotated by the hosted language implementer using Java reflection. We then compile these methods using the underlying Java virtual machine’s JIT compiler before runtime. Finally, we build a table that consists of the machine addresses of the compiled instruction operation methods. Algorithm 1 explains the algorithm used in the initialization phase, and Figure 3.8 visualizes the initialization phase in Jython.

Threaded Code Generation Phase

Next, we need to generate the actual subroutine threaded code. To generate threaded code, we need to map the hosted interpreter instructions to the native machine instructions. For instance, we map Python bytecode to the native machine instructions in the threaded gen-
**Algorithm 1 Initialization Phase**

1: procedure INITIALIZE_TABLE(class)
2: for method ∈ class.getDeclaredMethods() do
3: annotation ← method.getAnnotation()
4: if annotation ≠ ∅ then
5: opcode ← annotation.getAnnotationValue()
6: table[opcode] ← compile(method)
7: end if
8: end for
9: end procedure

---

**Algorithm 2**

---

---

**Figure 3.8:** Initialization phase of threaded code generator in Jython.

Algorithm 2 presents the algorithm at the heart of our threaded code generator. This algorithm contains several interesting details. First of all, we see that we need to decode the bytecode representation to find the opcode and operand argument of each instruction. Next, we see that our threaded code generator emits a `mov` instruction on line 7 to pass the operand argument to the instruction operation method. Then, we see how we use the table generated in the initialization phase to map the `opcode` to the actual machine code `address` on line 10. Finally, we see that our threaded code generator emits either a `call` instruction or a `jump` instruction on line 12 via `generateCallOrJump`. This is due to handling simple intra-procedural control flow, such as conditional and unconditional jumps. Context-threaded code includes inlining jump instructions into standard subroutine threaded code,
**Algorithm 2** Threaded Code Generation Phase

1: **procedure** `GENERATE_THREADED_CODE`(`bytecode`)  
2: \[\text{threaded code} \leftarrow \text{ALLOCATE}(\text{bytecode})\]  
3: \[\text{while } \text{bytecode} \neq \emptyset \text{ do}\]  
4: \[\text{opcode} \leftarrow \text{bytecode}.\text{NEXT_OPCODE}()\]  
5: \[\text{if } \text{opcode.HAVE_ARGUMENT()} \text{ then}\]  
6: \[\text{oparg} \leftarrow \text{bytecode}.\text{GET_OPARG}()\]  
7: \[\text{code} \leftarrow \text{GENERATE_MOVE_FOR_OPARG}(\text{oparg})\]  
8: \[\text{threaded code}.\text{APPEND}(\text{code})\]  
9: \[\text{end if}\]  
10: \[\text{compiled method} \leftarrow \text{table}[\text{opcode}]\]  
11: \[\text{address} \leftarrow \text{compiled method}.\text{GET_ENTRY_POINT}()\]  
12: \[\text{code} \leftarrow \text{GENERATE_CALL_OR_JUMP}(\text{opcode}, \text{address})\]  
13: \[\text{threaded code}.\text{APPEND}(\text{code})\]  
14: \[\text{end while}\]  
15: **end procedure**

too, but in contrast we do not perform its *tiny inlining* [23]. Figure 3.9 shows the threaded code generation phase in Jython.

![Figure 3.9: Threaded code generation phase in Jython.](image)

**Handling Operand Arguments**

Many bytecode implementations have two parts for each bytecode. For example, Python bytecode consists of one-byte *opcode* followed by zero or one *operand argument*. If a Python bytecode has an operand argument, then the two adjacent consecutive bytes are used to calculate its value. Operand arguments are similar to the *immediate operands* in Java bytecode.
Python bytecode includes many instructions which take an operand argument. For example, `LOAD_FAST` instruction is one of these instructions as shown in Figure 3.10. It pushes a reference onto the stack from a local variable at index specified by an operand argument. We extract the instructions that take an operand argument into their own methods, and pass the operand argument as an argument to these methods. We emit a `mov` machine instruction in the generated threaded code to pass the operand argument to the instruction operation method. Figure 3.11 shows the operand argument (`oparg`) handling in Jython for the `LOAD_FAST` instruction displayed in Figure 3.10.

![Figure 3.10: Operand argument in Python.](image1)

```java
PyObject interpret(PyFrame f, ...) {
    PyStack stack = new PyStack(stacksize);
    int next_instr;
    ...
    while (true) {
        int opcode = bytecode[next_instr++];
        int oparg;
        if (opcode >= Opcode.HAVE_ARGUMENT) {
            oparg = decodeOpArg(bytecode, next_instr);
            next_instr += 2;
        }
        ...
        switch (opcode) {
            ...
            case Opcode.LOAD_FAST:
                stack.push(f.getlocal(oparg));
                break;
            ...
            ...
        }
    }
    ...
}
```

Threaded code

```
mov rsi,$0x0
call &load_fast
```

![Figure 3.11: Handling operand arguments in Jython.](image2)
Handling Branch Instructions

Inlining unconditional branches is straightforward. We just translate an unconditional branch into a \texttt{jmp} machine instruction, thus eliminating the corresponding \texttt{call} instruction. To find the native machine jump target, we need to extract the virtual target address from the input bytecode instruction, and convert it to the corresponding target address at machine level while assembling the \texttt{jmp} instruction as illustrated in Figure 3.12.

![Python bytecode Threaded code]

Figure 3.12: Handling unconditional branch in Jython.

Inlining conditional branches is not directly possible, however. The key issue preventing this is that we cannot in general infer \texttt{Boolean} expression evaluation logic implemented in the interpreted hosted language. Therefore, we require the conditional branch instruction implementations to return a \texttt{Boolean} result.

Figure 3.13 shows the implementation of the \texttt{JUMP\_IF\_TRUE} conditional branch instruction in Jython. The generated threaded code includes a native machine \texttt{call} instruction to the associated instruction implementation, which evaluates the condition at runtime, and returns the result. Then, it performs a \texttt{test} instruction to set the flag based on the result of the condition. Finally, it includes a \texttt{jnz} conditional jump instruction to jump to the branch target.
Handling Return Instructions

Handling return instructions are straightforward as shown in Figure 3.14. The RETURN_VALUE instruction pops the value at the top of the stack, and returns it to the caller of the function in Jython. We first emit a native machine call instruction. Next, we deallocate the frame via addq instruction and emit a return instruction to return from the function.
Execution of Threaded Code

Algorithm 3 details the actual implementation of how all of the parts fit together. We lazily generate subroutine threaded code at runtime, in a just-in-time fashion, i.e., before its first execution. If our code cache is of limited size, we can easily remove previously generated, cached subroutine threaded code. Subsequent invocation triggers regeneration, which requires only a linear pass and therefore is efficient.

Algorithm 3 Execution of Threaded Code

1: procedure INTERPRET(method)
2: code ← GET_THREADED_CODE(method)
3: if code = ∅ then
4: code ← GENERATE_THREADED_CODE(method)
5: method.CACHE(code)
6: end if
7: CALL_THREADED_CODE(code)
8: end procedure

An Example

Figure 3.15 shows a simple Python function, add, that “adds” two local variables, item0, and item1 on the left. It presents the Python bytecode representation for the add function in the middle. Both of the LOAD_FAST instructions require an operand, zero and one respectively, to identify the corresponding local variables. On the right, it displays the threaded code we generate for the add function. We emit native machine call instructions to the associated instruction operation methods. Note that we inline the immediate operands zero and one from the LOAD_FAST instructions directly into the generated native machine code.
**Removing Array Store Checks**

Our current prototype implementation provides an annotation that acts as an intrinsic and instructs the just-in-time compiler to omit the `ArrayStoreException` check for the operand stack array. Java virtual machines should enforce certain restrictions before eliminating array store checks for the safety. They would use the dynamic deoptimization mechanism which aggressively eliminates the checks based on the annotation provided by the language implementer, and deoptimizes when the assumption fails.

Moreover, we should be able to verify that the interpreter only operates on objects of the same type similar to Java bytecode verification [62] We would need to implement the data-flow analysis described by Leroy to apply to all instruction operations and show that for all possible operands are bounded by some base class, such as `PyObject` in Jython. We did not, however, implement this step and leave this for future work.
Maxine VM

Modular VM extends the Maxine VM [89, 93]. Maxine VM is a research virtual machine for Java developed at Oracle Labs. Virtual machines are typically implemented in a lower level language. For instance, the two industrial-strength high-performance Java virtual machines, Oracle’s Java HotSpot virtual machine and IBM’s J9 virtual machine, are both written in C++. Whereas, Maxine is a metacircular Java virtual machine implemented entirely in Java.

Maxine has no interpreter, and has two compilers: T1X and C1X: T1X is template-based baseline compiler and is Maxine’s first line of execution. C1X is a new optimizing JIT compiler derived from porting the HotSpot client compiler [61]. Initially, T1X compiles a method. Whenever, it discovers a hot method, C1X compiles the method for producing better machine code by performing aggressive optimizations.

We use Maxine’s C1X compiler to compile the instruction operation methods in the initialization phase of our threaded generator. To emit the actual native machine code, we rely on Maxine’s backend.
Chapter 4

Efficient Profiler Framework

Optimizing an application to run as fast as possible on a given computing platform has always been a challenging task. An essential prerequisite for optimizing an application is to first understand its execution characteristics. A number of tools are available for the application developers to accomplish this, ranging from simple shell utilities, profilers, tracers, to sophisticated graphical tools.

Profilers are crucial for programmers to locate and eliminate performance bottlenecks towards the goals of optimization and performance tuning. Dynamic languages have great potential for performance improvement because they are typically slower than their static counterparts. Therefore, profilers play an important role in diagnosing performance issues in dynamic language applications.

Truffle [94, 54] is a new framework for creating high-performance implementations of hosted dynamic languages on a Java virtual machine. Language implementers write an AST interpreter, then the framework applies dynamic optimizations such as type specialization, and just-in-time compiles the AST to machine code. There are Truffle implementations for JavaScript, Python, Ruby, Smalltalk and R, which show superior performance compared to
existing implementations [52, 97, 51, 65, 57].

We develop a profiler framework [81] for the Truffle platform and evaluate it on two Truffle language implementations: ZipPy (Python implementation in Truffle)\(^1\) and JRuby+Truffle (Ruby implementation in Truffle)\(^2\). Our work focuses on how to build a generic and efficient profiler in the Truffle platform. We provide a comprehensive event-based profiler that profiles various events to analyze dynamic language programs in more detail. Our profiler compares and evaluates dynamic languages on cross-language benchmarks implementing the same algorithms.

### 4.1 Truffle Framework

Truffle is a self-optimizing runtime system for dynamic languages, where language implementations are expressed as AST interpreters in Java. A parser typically builds an AST, and the language implementer adds `execute` methods to the AST nodes to implement an AST interpreter. Every node in the AST has a list of child nodes. Each language function has a root node, and the execution starts by executing the root node. Every node calls `execute` methods of its child nodes to compute the result, and returns the result to its parent. Figure 4.1 shows a Python function that adds two variables, and the generated AST for this function on the left.

Dynamic languages provide many flexible features such as dynamic typing, higher level data structures, and continuations. In dynamic typing, type checks are performed during runtime and variables get the type of the object assigned to them at runtime. For example, “\(a + b\)” can perform integer addition, string concatenation, or a user-defined operation based on its input operands at runtime.

\(^1\)ZipPy profiler is available at [https://bitbucket.org/ssllab/zippy](https://bitbucket.org/ssllab/zippy)

\(^2\)JRuby+Truffle profiler is available at [https://github.com/gulfemsavrun/JRuby-Truffle-Profiler](https://github.com/gulfemsavrun/JRuby-Truffle-Profiler)
Figure 4.1: AST of an add function before and after inserting instrumentation nodes to the add operation event.

The Truffle framework optimizes dynamic typing by collecting type information, and speculatively replacing generic AST nodes with type-specialized nodes [95]. Typed AST nodes have multiple versions at runtime. They start with an uninitialized version. Upon the first execution of a node, Truffle chooses the first specialized version that matches the observed type, and rewrites the node to this specialization. When the current specialized version no longer matches with the new type on subsequent execution, Truffle replaces the node with another typed version or a generic version. This node-rewriting [31, 30] strategy relies on an important observation called type stability [40] that optimizes dynamic typing. The observation is that the operand types of an operation are unlikely to change during program execution. Therefore, Truffle speculatively rewrites generic nodes to type-specialized versions, and it replaces nodes back to generic nodes when speculation fails. Figure 4.2 shows an example
of possible multiple versions of an add node for a Truffle language implementation.

![Diagram of possible multiple versions of an add node for a Truffle language implementation.](image)

Figure 4.2: Typed versions of an add node in Truffle.

### 4.2 Instrumentation Framework

The Truffle platform includes multi-purpose instrumentation support for building program analyzers and other tools. An experimental built-in debugger in JRuby+Truffle demonstrated that extremely low runtime overheads could be achieved through tight integration with the underlying Truffle platform [84]. A generalized instrumentation framework is now in place, supporting development of a fully functional multi-language debugging service along with other tools such as code coverage. Here we focus on building an efficient profiler using Truffle instrumentation.

The instrumentation framework works by inserting additional AST nodes, essentially rewriting a program to include instrumentation functionality. Instrumentation nodes are no different from other Truffle AST nodes with respect to optimization, so inactive instrumentation nodes add zero overhead when running fully optimized. Instrumentation provided by client tools can be added and removed dynamically using core Truffle tree-rewriting mechanisms.

An AST node becomes instrumentable by insertion of a wrapper node between the node
and its parent. A wrapper node is language-specific, so it must be implemented separately by every language implementer. Each wrapper has an attached language-independent probe node, which represents the association with the particular piece of source code represented by the instrumentable node. The probe node also manages the attachment and detachment of tool-provided instrument nodes. When the execution reaches a wrapper node, it notifies the probe node both before and after executing its child; the probe node passes each of these execution events to every attached instrument node.

Figure 4.1 shows the AST after inserting a wrapper node and attaching an instrument node to the add node on the right. Figure 4.3 shows the implementation of instrumentation nodes.

4.3 Event-based Profiling

Event-based profilers track certain execution events to analyze the runtime behavior of a program. Counting AST nodes provides information about the execution of events in a program, so we count how many times certain AST nodes execute in our profiler. We divide collected events into five categories: control-flow, operation, collection, variable access, and type distribution.

**Control-flow:** Profiles statement nodes that change the control-flow such as loops, iterations, if, continue, next, and break statements. For example, this category shows a user how many times each loop or if statement executes.

**Operation:** Profiles operation nodes such as arithmetic, comparison, and logical operations in the program.

**Collection:** Profiles collections such as lists and arrays in the given program. Profiled collection events include reading an element from a collection, adding an element to a collection, deleting an element from a collection, and slicing a collection.
Variable access: Profiles nodes performing any kind of variable access, such as a local, global, and an instance variable access.

Type distribution: Profiles distribution of types of nodes in a program.

We insert a wrapper node to the node that we want to profile, and attach an event profiler instrument node to the wrapper node. The event profiler instrument node simply increments
the counter whenever the instrumented node executes as shown in Figure 4.4. We report how many times each node executes, and the source code location of the node in the program. We can simply extend our profiler to collect execution time of each event. Although we divide the collected events into five categories, the implementer can simply extend these categories based on her language.

```java
class EventProfilerInstrument extends Instrument {
    long counter;
    @Override
    void enter(Node astNode, VirtualFrame frame) {
        this.counter++;
    }
}
```

Figure 4.4: Implementation of an event profiler instrument node.

Dynamic typing adds a significant performance overhead to language execution. Truffle uses type-specialized nodes to reduce this overhead as explained in section 4.1. Demonstrating the distribution of types of nodes in a program is useful for two reasons. First, the language implementer can verify whether type-specializations are performed correctly in her implementation. Second, the language user could monitor the types in the program. In the example showed in Figure 4.1, the function executes two times: with integer arguments and string arguments, so the profiler reports two types for the add operation.

For type distribution profiling, we insert a wrapper node, and attach an instrument node that maps types to counters. Whenever a node executes, we check whether we have observed this type before. If this is a new type, we add it to the map. In ZipPy, type distribution profiling profiles the types for operations and variable accesses. It only profiles the types for variable accesses in JRuby+Truffle because Ruby represents operations as method calls.
4.4 Method Profiling

The reference implementations of dynamic languages typically provide method profilers that measure the number of method invocations and the time spent in each method. Both CPython [77] and MRI (CRuby) [66], Python and Ruby reference implementations respectively, have method profilers.

Python has a profiler called `cProfile` [78] implemented as a C extension module. It is an event-based profiler which records all method calls and returns, and measures time intervals between these events. Another Python profiler is `profile`. It is a pure Python module that has the same interface as `cProfile`. The disadvantage of the `profile` module is that it adds significant overhead to profiled programs.

Ruby has a `profile` library that is part of the standard library. Since it is a pure Ruby library, profiled programs run slow. An alternative profiler is `ruby-prof` [2] which is a C extension and therefore is many times faster than the standard Ruby profiler. JRuby [5] is the implementation of Ruby on top of the Java virtual machine (JVM). It uses the new `invokedynamic` bytecode [87], which improves the costly invocation semantics of dynamic programming languages targeting the JVM. JRuby has a built-in profiler [6], which is a clone of `ruby-prof` built into JRuby. It produces similar output to `ruby-prof`, and users need to enable the `–profile` flag to use it.

Since detecting frequently executed methods is a useful feature, we also add this functionality to our profiler. We insert a wrapper node to call nodes and attach an instrument node that collects invocation counter and time. We record the time before and after a call node executes and calculate the elapsed time. Figure 4.4 shows the implementation of our method profiler instrument.

In ZipPy, our method profiler output is similar to `cProfile` for Python. We show the total
number of calls and execution time for each user-defined and built-in function in the running application. We show the total time spent in each function by excluding time made in calls to sub-functions, and cumulative time spent in this and all sub-functions.

Ruby represents many of its internals as methods calls such as operators and property accesses. For example, Ruby translates “a + b” statement into “a.+(b)”, which is a call to a method named “+”. Our method profiler in JRuby+Truffle produces output similar to the built-in profiler in JRuby. We show the total time spent in each method, and break the total time into self and children time. Self time shows the time spent in the method itself, excluding calls to child methods. Children time shows the time spent in calls to the child methods.

### 4.5 Cross-language Comparison

Existing profilers detect performance bottlenecks in a program written in a specific language. For example, a Python profiler analyzes a Python program, and identifies performance bottlenecks.
necks in that program. Truffle focuses on implementing high-performance implementations of multiple dynamic languages. Implementing a profiler on this framework lets us compare programs across different languages. Languages have their unique features, and using these unique features results in executing different events. Profiling different languages on cross-language benchmarks that implement the same algorithms allows us to show common and different features in languages. To the best of our knowledge, our profiler is the first profiler that compares and analyzes the dynamic language programs for the same algorithms.
Chapter 5

Evaluation

5.1 Efficient Hosted Interpreters

In this section, we evaluate the performance potential of our optimizations using three popular dynamic language implementations targeting the Java virtual machine: Jython [7], Rhino [12], and JRuby [5], which implement Python, JavaScript, and Ruby, respectively. We start by explaining our system setup. Then, we show our benchmark results for Jython, Rhino, and JRuby and analyze the effectiveness of subroutine threaded code and array store optimization. Furthermore, we analyze the performance effect of using the HotSpot server compiler. Next, we present our results of investigating the implementation effort required by implementing a hosted interpreter and a host VM targeted compiler. Finally, we compare the frequency of array store checks between regular Java programs and our subroutine threaded code interpreter.
5.1.1 Experimental Setup

Interpreters. Both Jython and Rhino have a bytecode interpreter, therefore, we use their existing bytecode interpreters in our evaluation. On the other hand, JRuby does not have a bytecode interpreter, so we port the YARV (yet another RubyVM) [80] interpreter, which is the bytecode interpreter of Ruby in C, into Java. These three language implementations implement their own host VM targeted compilers, which compile their corresponding input languages directly down to Java bytecode. We use the following configuration for the dynamic language implementations:

- Jython 2.7.0a2.
- Rhino 1.7R4.
- JRuby 1.7.3.

Benchmarks We select several benchmarks from the computer language benchmarks game [47], a popular benchmark suite for evaluating the performance of different programming languages. We use the following benchmarks to measure the performance of our modified systems in Jython, Rhino, and JRuby:

- binarytrees: Recursive calls to allocate and deallocate binary trees
- fannkuchredux: Repeatedly access a tiny integer-sequence
- fasta: Generate and write random DNA sequences
- mandelbrot: Generate a Mandelbrot set and write a portable bitmap
- meteor: Search for solutions to shape packing puzzle (not only available for Rhino)
- nbody: Perform an N-body simulation of the Jovian planets
• **spectralnorm**: Calculate an eigenvalue using the power method

**Java virtual machines** Our Java virtual machine configuration is as follows:

- Maxine build from revision number 8541 (committed on October 16th, 2012)
- Oracle’s HotSpot Java virtual machine version 1.6

**Procedure** We use Intel Xeon E5-2660 based system, running at a frequency of 2.20 GHz, using the Linux 3.2.0-29 kernel and gcc version 4.6.3. We run each benchmark with multiple arguments to increase the range of measured running time. We run ten repetitions of each benchmark with each argument and report the geometric mean over all runs.

### 5.1.2 Comparison with Switch-based Interpreter

Table 5.1 shows the average speedups of our interpreter optimizations over Jython’s, Rhino’s and JRuby’s switch-based interpreters, respectively. We normalize the performance by the switch-based interpreter. Figure 5.4 reports our speedups for each benchmark.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Threaded Code</th>
<th>Efficient Array Stores</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jython</td>
<td>1.73×</td>
<td>1.42×</td>
<td>2.45×</td>
</tr>
<tr>
<td>Rhino</td>
<td>2.13×</td>
<td>1.68×</td>
<td>3.57×</td>
</tr>
<tr>
<td>JRuby</td>
<td>1.64×</td>
<td>1.54×</td>
<td>2.52×</td>
</tr>
</tbody>
</table>

Table 5.1: Performance of subroutine threaded code interpreter relative to switch-based interpreters.

We achieve a 1.73× speedup over the switch-based interpreter only from subroutine threaded code in Jython. Similarly, we achieve a 2.13× speedup from subroutine threaded code in Rhino. Furthermore, we gain a 1.64× speedup over the switch-based interpreter by applying subroutine threaded code in JRuby.
Figure 5.1: Jython.

Figure 5.2: Rhino.

Figure 5.3: JRuby.

Figure 5.4: Performance of subroutine threaded code interpreter relative to switch-based interpreters.
The average speedup of 2.13× over Rhino’s switch-based interpreter is higher than the average speedups from Jython’s and JRuby’s interpreter from subroutine threaded code. The reason is that Rhino executes more instructions for the same benchmarks, so the dispatch overhead is higher in Rhino. Therefore, reducing the dispatch overhead by subroutine threaded code gives higher speedups in Rhino.

**Efficient Array Store Performance**

We gain an additional 1.42× speedup by applying efficient array stores to our subroutine threaded code interpreter for Jython. With the efficient array stores, we achieve a 2.45× speedup over the switch-based interpreter in Jython. Similarly, we achieve an extra 1.68× speedup by applying the same optimization to our subroutine threaded code interpreter for Rhino. When combined with the efficient array stores, we achieve a 3.57× speedup over the switch-based interpreter in Rhino. Moreover, we gain an additional 1.54× speedup from efficient array stores in JRuby. Together with the efficient array stores, we achieve a 2.52× speedup over the switch-based interpreter in JRuby.

We use the `perf` [8] tool to measure the number of native machine instructions eliminated by this optimization. We find out that array store optimization removes 35% of the executed native machine instructions on average in our subroutine threaded code interpreter for Jython. Similarly, it reduces the executed native machine instructions by 44% on average in our subroutine threaded code interpreter for Rhino. Furthermore, it removes 31% of the executed native machine instructions on average in our subroutine threaded code interpreter for JRuby.

The spectrum of the speedups is relatively wide. Our subroutine threaded code interpreter and efficient array stores together performs better in the benchmarks that have higher dispatch overhead, such as `fannkuchredux`. For example, the Python version of this benchmark
only has one Python function, therefore, the interpreter is only invoked once. Currently, Maxine does not support on-stack replacement which allows the VM to replace the stack frame with that of an optimized version. Therefore, Maxine never gets a chance to recompile the switch-based interpreter for fannkuchredux.

When C1X recompiles the switch-based interpreter, it produces better code, so the switch-based interpreter performs better. For instance, the subroutine threaded code interpreter speedups are lower for the call-intensive binarytrees benchmark.

5.1.3 Comparison with Host VM Targeted Compiler

Table 5.2 shows the average performance of our interpreter optimizations compared to Jython’s, Rhino’s, and JRuby’s host VM targeted compilers, respectively. Figure 5.8 reports our performance for each benchmark. We normalize the performance by the host VM targeted compiler performance, i.e., values lower than 1.0 indicate a slow-down relative to the host VM targeted compiler.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Threaded Code</th>
<th>Threaded Code + Efficient Array Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jython</td>
<td>0.70×</td>
<td>0.99×</td>
</tr>
<tr>
<td>Rhino</td>
<td>0.42×</td>
<td>0.72×</td>
</tr>
<tr>
<td>JRuby</td>
<td>0.36×</td>
<td>0.58×</td>
</tr>
</tbody>
</table>

Table 5.2: Performance of subroutine threaded code interpreter relative to host VM targeted compilers.

For Jython, subroutine threaded code achieves 0.70× of the performance of the host VM targeted compiler. Together with the efficient array stores, it delivers 0.99× of the performance of Jython’s host VM targeted compiler. Likewise, subroutine threaded code itself delivers 0.42× of the performance of the host VM targeted compiler in Rhino. In combination with the efficient array stores, the subroutine threaded interpreter achieves 0.72× of the perfor-
Figure 5.5: Jython.

Figure 5.6: Rhino.

Figure 5.7: JRuby.

Figure 5.8: Performance of subroutine threaded code interpreter relative to host VM targeted compilers.
mance of Rhino’s host VM targeted compiler. Moreover, subroutine threaded code brings 0.36× of the performance of the host VM targeted compiler in JRuby. When combined with the efficient array stores, it delivers 0.58× of the performance of the host VM targeted compiler.

Our average performance of 0.70× and 0.58× compared to Rhino’s and JRuby’s host VM targeted are lower than our average performance of 0.99× compared to Jython’s host VM targeted compiler. The reason is that Rhino and JRuby have a more complex compiler implementing aggressive optimizations, such as data flow and type inference analyses. For example, Rhino’s host VM targeted compiler uses nine different optimization levels, and we use the highest optimization level in our evaluation.

Our subroutine threaded code interpreter with efficient array stores outperforms the host VM targeted compiler in fannkuchredux, fasta, mandelbrot, and nbody benchmarks in Jython. We report the two highest speedups in fannkuchredux, and mandelbrot.

Jython’s host VM targeted compiler compiles each Python program into one class file, and generates a Java method for each Python function. Maxine initially compiles each Java method using its template-based just-in-time compiler, T1X. When a method becomes hot, Maxine recompiles it using its optimizing just-in-time compiler, C1X. Hence, subsequent invocations of this method execute the optimized code. However, the Java method generated by Jython’s host VM targeted compiler must be invoked at least more than a certain threshold number of times to trigger the recompilation by C1X. Fannkuchredux and mandelbrot have only one Python function that executes a hot loop. Without on-stack replacement, Maxine is not able to produce optimized code for these two benchmarks. As a result, the host VM targeted compiler does not perform well for these benchmarks.

For call intensive benchmarks, such as binarytrees, our optimized interpreter performs worse than the host VM targeted compiler. The optimizing JIT compiler is able to recompile the
Java methods generated by the host VM targeted compiler at a very early stage in this benchmark.

5.1.4 Comparison with HotSpot Server Compiler

Previous studies of threaded code used a traditional, ahead-of-time compiler which performs many time-consuming optimizations. In contrast, Maxine’s C1X compiler—and the compiler it is modeled after, HotSpot’s client compiler C1 [61]—focuses on keeping compilation time predictably low by omitting overly time-consuming optimizations. On the other hand, HotSpot’s server compiler—known as C2 [76]—generates higher-quality code at the expense of higher latency imposed by longer compilation times.

To qualify the potential gain from using a more aggressive compiler, we compare the impact of our optimizations to compiling Jython’s switch-based interpreter with the server compiler. Figure 5.12 shows the speedups of compiling Jython’s switch-based interpreter with the Java Hotspot’s server compiler. We normalize the performance by the client compiler. We find that the server compiler achieves an average 1.81× speedup for Jython. Similarly, it achieves an average 1.10× speedup for Rhino. Moreover, the server compiler delivers an average 1.09× speedup for JRuby. Therefore, while using an aggressively optimizing compiler does give a better baseline to the interpreter, it does not offset our performance gains, but puts them into perspective with the reported speedup potential in the relevant literature [46].

Furthermore, our technique allows the fast client compiler to outperform the server compiler without using any of the more expensive optimization techniques, which certainly has practical implications, for example in embedded systems or smartphones, where energy-efficiency is key.
Figure 5.9: Jython.

Figure 5.10: Rhino.

Figure 5.11: JRuby.

Figure 5.12: Performance of compiling switch-based interpreters with Java Hotspot’s server compiler relative to client compiler.
5.1.5 Implementation Effort Comparison

Hosted Interpreter vs. Host VM Targeted Compiler Implementation Effort

We found that some implementations targeting the Java virtual machine started out by porting their C implementation counterparts to Java. Due to the bottlenecks identified in chapter 3, performance-conscious language implementers will invariably write a host VM targeted compiler. This investment, however, is costly in terms of initial implementation and continuous maintenance efforts.

Comparing the number of lines of code in an interpreter and a host VM targeted compiler gives an indication of the complexity of a host VM targeted compiler. Therefore, we counted the number of lines of code in a hosted interpreter and a host VM targeted compiler for three programming language implementations. We measure the number of Java lines of code for Jython, Rhino and JRuby. Table 5.3 reports the lines of code numbers for each of these programming language implementations. The second and fourth columns list the files or package names counted. The third and fifth columns show the number of lines counted from these packages. The last column shows the factor of the code size reduction between the hosted interpreter and host VM targeted compiler of a particular implementation.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Interpreter File</th>
<th># Lines</th>
<th>Host VM Targeted Compiler Package</th>
<th># Lines</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jython</td>
<td>org.jython.core.PyBytecode.java</td>
<td>1097</td>
<td>org.jython.compiler</td>
<td>5007</td>
<td>∼ 5×</td>
</tr>
<tr>
<td>Rhino</td>
<td>org.mozilla.classfile</td>
<td>4183</td>
<td>org.mozilla.javascript.optimizer</td>
<td>5790</td>
<td>∼ 4×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>total</td>
<td>9973</td>
<td></td>
</tr>
<tr>
<td>JRuby</td>
<td>org.jruby.ast.executable.YARVMachine.java</td>
<td>1346</td>
<td>org.jruby.compiler</td>
<td>5995</td>
<td>∼ 9×</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>org.jruby.compiler.impl</td>
<td>6744</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>org.jruby.compiler.util</td>
<td>339</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>total</td>
<td>12178</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Hosted interpreter vs. host VM targeted compiler implementation effort.

Jython’s host VM targeted compiler has 5007 lines of Java code. This line count does not include the ASM Java bytecode framework [29] used by the compiler for bytecode assembling. Jython’s host VM targeted compiler relies extensively on generating JVM-level calls to its
runtime system to simplify the compilation process. This technique results in a relatively small and manageable compiler. Rhino has a more complicated, optimizing compiler consisting of 9973 lines of Java code. Similarly, JRuby has a more complex compiler including 12178 lines of Java code.

The Hosted Language Implementer’s Implementation Effort

As explained in subsection 3.3.1, the language implementer only needs to add two new lines of code for each i-op to the existing switch-based interpreter to enable threaded code generation. For example, this manual transformation results in \( \sim 250 \) new lines of Java code in Jython, and \( \sim 140 \) new lines of Java code in Rhino, and \( \sim 160 \) new lines of Java code in JRuby.

The Java Virtual Machine Implementer’s Implementation Effort

On the other hand, the threaded code generator for Jython requires only 337 lines of Java code. Similarly, the threaded code generator for Rhino needs 393 lines of Java code, and JRuby requires 317 lines of Java code.

5.1.6 Array Store Check Frequency

As mentioned in section 3.2, we find that efficient array stores are crucial for the performance of the interpreters. Interpreters written in Java are atypical since they perform significantly more of array store checks. To show that array store checks in interpreters are more frequent than regular Java programs, we compare the DaCapo benchmarks [25] to our subroutine threaded code interpreter for Jython. We select several benchmarks from the DaCapo benchmark suite. Table 5.4 and Table 5.5 show the measurements from the DaCapo benchmarks.
and Jython’s subroutine threaded code interpreter. The first column shows the total number of executed native machine instructions and the second column reports the total number of array store checks performed in millions for each specific benchmark. The last column shows the frequency of the array store checks per million native machine instructions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Native Machine Instructions</th>
<th># Array Store Checks</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>42,598</td>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>eclipse</td>
<td>433,621</td>
<td>16</td>
<td>39</td>
</tr>
<tr>
<td>fop</td>
<td>37,252</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>h2</td>
<td>230,879</td>
<td>47</td>
<td>204</td>
</tr>
<tr>
<td>lusindex</td>
<td>24,730</td>
<td>2</td>
<td>86</td>
</tr>
<tr>
<td>lusearch</td>
<td>82,630</td>
<td>7</td>
<td>88</td>
</tr>
<tr>
<td>pmd</td>
<td>124,284</td>
<td>17</td>
<td>137</td>
</tr>
<tr>
<td>sunflow</td>
<td>119,884</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>tomcat</td>
<td>90,257</td>
<td>12</td>
<td>135</td>
</tr>
<tr>
<td>xalan</td>
<td>92,174</td>
<td>10</td>
<td>109</td>
</tr>
</tbody>
</table>

Table 5.4: Array store check frequencies in millions for the DaCapo benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Native Machine Instructions</th>
<th># Array Store Checks</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 16</td>
<td>366,053</td>
<td>1,039</td>
<td>2,840</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>840,451</td>
<td>1,772</td>
<td>2,110</td>
</tr>
<tr>
<td>fasta 2,500,000</td>
<td>184,670</td>
<td>280</td>
<td>1,522</td>
</tr>
<tr>
<td>mandelbrot 1,000</td>
<td>168,844</td>
<td>373</td>
<td>2,217</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>205,587</td>
<td>337</td>
<td>1,639</td>
</tr>
<tr>
<td>nbody 500,000</td>
<td>328,593</td>
<td>924</td>
<td>2,809</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>488,248</td>
<td>1,485</td>
<td>3,040</td>
</tr>
</tbody>
</table>

Table 5.5: Array store check frequency in millions for Jython’s subroutine threaded code interpreter.

The average frequency of the array store checks is 97 per million native machine instructions in the DaCapo benchmarks whereas it is 2,311 per million native machine instructions in our subroutine threaded code interpreter. Therefore, the frequency of the array store checks is $\sim 24 \times$ that of typical Java programs. The high frequency of array store checks in interpreters means that array store optimizations are particularly effective at improving the performance.
of hosted interpreters.

5.1.7 Discussion

Unsurprisingly, comparing against the optimized vs. switch-dispatch virtual machine interpreters of both Jython, Rhino and JRuby shows significant speedups. It is worth noting, however, that half of the speedup is due to eliminating the expensive ArrayStoreException check. Interestingly, for fannkuchredux on Rhino, the dramatic speedup is due to having efficient array stores. This is since fannkuchredux executes the most hosted virtual machine instructions and almost all instructions include an array store check before writing their result to the operand stack.

Relatively to host VM targeted compilers, we find that a Java virtual machine implementing our optimizations makes the price/performance ratio of interpretation even more attractive: with little manual modification, an efficient hosted JVM interpreter is competitive with a simple host VM targeted compiler. Furthermore, we notice that the performance impact of eliminating the ArrayStoreException check changes noticeably. Compared with host VM targeted compiler performance, it does not contribute half, but roughly a third of the reported speedups.

We investigate whether the evaluated host VM targeted compilers use an array to pass operands. Rhino’s and JRuby’s host VM targeted compilers map the local variables in the hosted language to Java locals. Therefore, they operate on Java local variables and don’t use arrays as operand stacks. On the other hand, Jython’s host VM targeted compiler does not map Python variables to Java local variables. It generates virtual calls to Jython’s runtime methods to perform operations. If the method returns a result, it pushes its result onto the Java operand stack. So, Jython’s host VM targeted compiler uses Java operand stack to pass operands instead of implementing a separate operand stack. As a result, host VM targeted
compilers do not necessarily benefit from array store optimization while interpreters gain substantial speedups from it.

Concerning the implementation effort, our evaluation led to the following insights. First, using annotations for enabling our optimizations requires minimal effort by the hosted language implementer; this supports our claim that we can measure the effort in a matter of hours. Second, our investigation of the implementation effort for the Java virtual machine implementer shows that the threaded code generators mostly diverge in supporting separate bytecode decoding mechanisms. Most of the bytecode decoding logic can be factored into separate annotations which is why we think that supporting a large set of different decoding mechanisms is largely an engineering problem.

We present both the hosted language implementer and the JVM language implementer perspective in our implementation. When our dispatch optimizations can be applied automatically, the features of threaded code generator need not be exposed to the hosted language implementer. Therefore, regular Java programmers should not be able to access to the parts of the threaded code generator that might cause security holes in the JVM. On the other hand, our array store elimination might break the JVM code if used in inappropriate places. We can add an additional verification to make it more secure. However, JVMs already provide unsafe features such as \texttt{sun.misc.Unsafe} and JNI that can also destabilize the VM.
5.2 Efficient Profiler Framework

In this section, we present a detailed evaluation of our profiler in ZipPy and JRuby+Truffle. We begin by explaining our system setup. Next, we show our benchmark results for our event-based profiling and method profiling. Furthermore, we present our results of cross-language comparison. Finally, we compare the implementation effort required by our profiler and other existing profilers.

5.2.1 Experimental Setup

Benchmarks We include all the benchmarks that are available in ZipPy and JRuby+Truffle implementations. There are six benchmarks (binarytrees, fannkuchredux, mandelbrot, nbody, pidigits, spectralnorm) from the Computer Language Benchmarks Game [47], and one benchmark (richards) from the V8 benchmark suite [48], which was originally developed for BCPL.

- **binarytrees**: recursive calls to allocate and deallocate binary trees
- **fannkuchredux**: repeatedly access a tiny integer-sequence
- **mandelbrot**: generates a Mandelbrot set
- **nbody**: perform an N-body simulation of the Jovian planets
- **pidigits**: streams arbitrary-precision arithmetic
- **richards**: performs an OS kernel simulation
- **spectralnorm**: calculate an eigenvalue using the power method

System configuration Our system configuration is as follows:
• CPython 3.3.2.
• PyPy 3.2.1.
• MRI (CRuby) 1.9.3.
• JRuby 1.7.13.
• Truffle 0.5.
• Intel Xeon E5-2660 running at a frequency of 2.20 GHz, the Ubuntu Linux 3.2.0-64 kernel and gcc 4.6.3.

We execute each benchmark ten times and report individual and average execution times.

5.2.2 Event-based Profiling Performance

Figure 5.13 demonstrates the contribution of each event counting category to the total execution time in ZipPy. Control-flow, operation, collection, variable access, and type distribution profiling add an average 4%, 7%, 3%, 25%, 34% overhead in ZipPy. Control-flow, operation, and collection profiling add a low overhead in ZipPy. However, variable access and type distribution profiling add a higher overhead. For example, *fannkuchredux* benchmark performs a significant number of variable accesses, so variable access profiling adds a significant overhead in that benchmark. Type distribution profiling collects types for operations and variable accesses, therefore, it also adds a high overhead in *fannkuchredux* benchmark.

Figure 5.14 demonstrates the contribution of each event counting category to the total execution time in JRuby+Truffle. Control-flow, operation, collection operation, variable access, and type distribution profiling add an average 4%, 13%, 4%, 41%, 45% overhead in JRuby+Truffle. Similar to ZipPy, control-flow, operation and collection operation profiling
Figure 5.13: Event-based profiling in ZipPy.

Figure 5.14: Event-based profiling in JRuby+Truffle.

61
only add a small overhead, but variable access and type distribution profiling add a higher overhead in JRuby+Truffle.

Control-flow profiling is especially useful for language users because they can observe which loops or statements are hot in their program. Statement-granularity profiling might be more insightful than method-granularity profiling to the programmers in some cases.

Collection profiling is also useful for language users. Python and Ruby each provide a rich set of collections, and users heavily use them in their programs. To the best of our knowledge, there is no Python or Ruby profiler that provides information about collection usage. Therefore, our unique collection profiling gives hints to users about how to optimize their collection usage. For example, tuples are immutable collections and lists are mutable collections in Python. By looking at the collection profiling output, users might learn that they never modify the list, therefore, they can use a tuple instead. As future work, we plan to give feedback and suggestions about collections based on the collected profiling information in our profiler.

Type distribution is an important category for both users and implementers. When the types are not stable in a given program, Truffle is not able to optimize that program well. Our type distribution profiling draws users attention to the nodes that frequently change their type during execution. Then, the user might modify their program for better performance.

Our event-based profiler is not limited to the event categories shown here. It is extensible, so the profiler implementer can add more events to capture her language’s characteristics.

5.2.3 Method Profiling Performance

Figure 5.15 shows method profiling performance across different Python implementations. We compare our method profiler in ZipPy against CPython [77] and PyPy [9, 26] using
cProfile module to profile methods. CPython is the reference Python implementation that has an interpreter written in C. PyPy is the Python implementation written in a subset of Python, and uses a tracing JIT compiler to optimize frequently executed parts of the program [26]. The x axis labels benchmarks, and y axis displays the execution time in a logarithmic scale.

In ZipPy, our profiler runs $1.1 \times$ faster than PyPy and $10 \times$ faster than CPython on average while using cProfile module. PyPy performs slightly better than ZipPy in nbody and pidigits benchmarks without profiling. When profiling is enabled, ZipPy performs better than PyPy in binarytrees, fannkuchredux, mandelbrot and richards, and PyPy performs better than ZipPy in spectralnorm. Among our benchmarks, binarytrees, mandelbrot, richards, and spectralnorm perform the highest number of calls as displayed in Table 5.6. As a result, profiling methods adds a higher overhead in these four benchmarks.

PyPy is a fast alternative implementation of Python that uses a tracing JIT compiler for producing optimized code, however, it is not able to perform well when profiling is enabled. The reason is that cProfile is a module written in C, and PyPy is not able to optimize C extension modules. The PyPy community states that they support C extension modules just to provide basic functionality. They advise users to use a native Python implementation instead of a C module for better performance. We also use profile module which is a pure Python module to profile our benchmarks in PyPy. However, PyPy performs worse with this module, so we only include the numbers from PyPy while using the cProfile module.

Figure 5.16 shows method profiling performance among different Ruby implementations. We compare our method profiler against MRI (CRuby) using ruby-prof and JRuby’s built-in profiler. We report an average speedup of $12 \times$ over JRuby’s built-in profiler and an average speedup of $208 \times$ over MRI profiler. With and without profiling, JRuby+Truffle performs better than JRuby and MRI in all the benchmarks.
Figure 5.15: Method profiling comparisons for Python.

Figure 5.16: Method profiling comparisons for Ruby.
As mentioned before, Ruby represents many of its internals as method calls. Similarly, JRuby and JRuby+Truffle specializes certain basic operation calls, and does not profile them. Whereas, MRI does not specialize certain basic operation calls.

We verify the results of our method profilers by comparing the number of method invocations across different profilers. We find out that our method profilers give the same results for the number of method invocations as the profilers that we compare them against.

### 5.2.4 Cross-language Comparison

Our profiler helps to compare and evaluate the programs implementing the same algorithms written in different languages. As an experiment, we compare and evaluate two different languages, Python and Ruby, on cross-language benchmarks implementing the same algorithms with our profiler. Table 5.6 and Table 5.7 list the total number of executed events in ZipPy and JRuby+Truffle in various categories. They execute different numbers of events in each profiling category for the same benchmarks. We highlight the categories that show significantly different results in two languages on cross-language benchmarks.

Python and Ruby have their unique features, and using these unique features results in executing different numbers of nodes. For example, `mandelbrot` operates on complex numbers. Python has a built-in complex type, whereas Ruby does not provide a built-in complex type. Therefore, Ruby performs many more operations to do complex number arithmetic.

Python has a built-in method called `abs` which operates on complex numbers, and `mandelbrot` intensively uses this function, so it performs more methods calls than Ruby. In `nbody`, Ruby defines a class and creates objects for every Jovian planet, whereas Python uses a collection to hold Jovian planets. Therefore, Ruby performs many instance method calls, and Python performs more collection operations in `nbody`. As a result, our profiler makes it possible to compare and analyze the programs implementing the same algorithms across different
languages.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Call</th>
<th>Control-flow</th>
<th>Operation</th>
<th>Collection</th>
<th>Variable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>329 × 10^6</td>
<td>495 × 10^6</td>
<td>410 × 10^6</td>
<td>1</td>
<td>2 × 10^9</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>739</td>
<td>12 × 10^9</td>
<td>13 × 10^9</td>
<td>9 × 10^9</td>
<td>55 × 10^9</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>416 × 10^6</td>
<td>905 × 10^6</td>
<td>1 × 10^9</td>
<td>1</td>
<td>2 × 10^9</td>
</tr>
<tr>
<td>nbody</td>
<td>387</td>
<td>116 × 10^6</td>
<td>1 × 10^9</td>
<td>1 × 10^9</td>
<td>3 × 10^9</td>
</tr>
<tr>
<td>pidigits</td>
<td>1 × 10^9</td>
<td>8 × 10^9</td>
<td>27 × 10^9</td>
<td>0</td>
<td>61 × 10^9</td>
</tr>
<tr>
<td>richards</td>
<td>1 × 10^9</td>
<td>3 × 10^9</td>
<td>2 × 10^9</td>
<td>165 × 10^6</td>
<td>11 × 10^9</td>
</tr>
<tr>
<td>spectralnorm</td>
<td>470 × 10^6</td>
<td>481 × 10^6</td>
<td>5 × 10^9</td>
<td>480 × 10^6</td>
<td>8 × 10^9</td>
</tr>
</tbody>
</table>

Table 5.6: Total number of executed events in ZipPy.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Call</th>
<th>Control-flow</th>
<th>Operation</th>
<th>Collection</th>
<th>Variable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>658 × 10^6</td>
<td>495 × 10^6</td>
<td>575 × 10^6</td>
<td>0</td>
<td>2 × 10^9</td>
</tr>
<tr>
<td>fannkuchredux</td>
<td>216</td>
<td>5 × 10^9</td>
<td>10 × 10^9</td>
<td>10 × 10^9</td>
<td>53 × 10^9</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>34</td>
<td>920 × 10^6</td>
<td>4 × 10^9</td>
<td>0</td>
<td>9 × 10^9</td>
</tr>
<tr>
<td>nbody</td>
<td>573 × 10^6</td>
<td>116 × 10^6</td>
<td>2 × 10^9</td>
<td>109 × 10^6</td>
<td>6 × 10^9</td>
</tr>
<tr>
<td>pidigits</td>
<td>1 × 10^9</td>
<td>8 × 10^9</td>
<td>27 × 10^9</td>
<td>0</td>
<td>68 × 10^9</td>
</tr>
<tr>
<td>richards</td>
<td>2 × 10^9</td>
<td>2 × 10^9</td>
<td>761 × 10^6</td>
<td>29 × 10^9</td>
<td>5 × 10^9</td>
</tr>
<tr>
<td>spectralnorm</td>
<td>932 × 10^6</td>
<td>481 × 10^6</td>
<td>4 × 10^9</td>
<td>462 × 10^6</td>
<td>7 × 10^9</td>
</tr>
</tbody>
</table>

Table 5.7: Total number of executed events in JRuby+Truffle.

5.2.5 Profiler Implementation Effort Comparison

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>1073</td>
</tr>
<tr>
<td>ZipPy</td>
<td>508</td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td>516</td>
</tr>
<tr>
<td>Python cProfile profiler</td>
<td>1650</td>
</tr>
<tr>
<td>JRuby built-in profiler</td>
<td>2223</td>
</tr>
</tbody>
</table>

Table 5.8: Profiler implementation effort comparison.

We implement and evaluate our profiler in ZipPy and JRuby+Truffle, but our profiler is reusable among other implementations of Truffle such as JavaScript, Smalltalk, and R. Table 5.8 lists the number of lines of generic profiler framework code shared by ZipPy...
and JRuby+Truffle, and language-dependent code implemented separately in ZipPy and JRuby+Truffle. Although language-dependent parts are similar in ZipPy and JRuby+Truffle, the language implementers must implement them separately for their own language. It also shows the number of lines of code used to implement Python cProfile profiler and JRuby’s built-in profiler. As a result, we implement a generic profiler framework with modest implementation effort.
Chapter 6

Related Work

6.1 Threaded Code

In 1973, Bell [21] introduces threaded code as an intermediate point between interpretive code and machine code. He uses FORTRAN IV compiler to generate threaded code for FORTRAN programs on a PDP-11 system. Bell finds out that threaded code is approximately as fast as machine code, and requires less space than machine code in his evaluation.

In 1975, Dewar [41] introduces indirect threaded code, and starts to refer Bell’s threaded code as direct threaded code. He implements indirect threaded code for SNOBOL4 system. SNOBOL (StriNg Oriented and symBOlic Language) is a series of programming languages developed by AT&T Bell Laboratories between 1962 and 1967. Dewar compares his indirect threaded code to Bell’s direct threaded code generated by the PDF-11 FORTRAN compiler. He reports that his implementation is less machine-dependent, requires less space, and executes in fewer cycles.

In 1982, Kogge [60] describes the advantages of using threaded code and systematically
analyses its performance potential. In 1982, Thanh and Raschner [88] explain their implementation of changing the Pascal compiler to generate indirect threaded code. They compare their implementation to two other existing Pascal systems, Concurrent Pascal designed by Per Brinch Hansen and UCSD Pascal developed at the University of California, San Diego.

In 1990, a book by Debaere and van Campenhout [39] reports the state-of-the-art for threaded code, including an in-depth discussion of all threaded code techniques known at the time, plus the effects of using a dedicated co-processor to “offload” the interpretative overhead from the critical path of the CPU. In 1993, Curley [37, 38] explains the subroutine threaded code for Forth running on the 68000 CPU, and Ertl [42] describes the direct and indirect threaded code for their Forth system implemented in GNU C.

In 1996, Romer et al. [79] examine the performance of MIPSI, Java, Perl, and Tcl interpreters running on a DEC Alpha platform. They analyze the effect of software and hardware on the interpreter performance. The paper shows that the dispatch loop is not the only performance bottleneck in an interpreter. There are other software factors such as the complexity of the virtual instruction set and the use native runtime libraries. Moreover, the authors claim that the performance of interpreters would not benefit much from the hardware components, such as the branch predictor and translation lookaside buffer (TLB). As a result, implementers should focus on software aspect instead of hardware aspect to improve the performance of their interpreter.

Ertl and Gregg [44, 46] investigate the performance of the interpreters designed for high performance. They study four efficient interpreters, Gforth, OCaml, Scheme48, and Yap, and use two inefficient interpreters, Perl and Xlisp, for comparison purposes. The interpreters run on the SimpleScalar microprocessor simulator, simulating a close derivative of the MIPS architecture. The authors find out that efficient interpreters perform many indirect branches and using threaded code for instruction dispatch makes them up to twice as fast as interpreters using switch-based dispatch. They simulate various branch predictors, and analyze
their affect on the interpreter performance.

In 2003, Ertl and Gregg [45] discuss the affect of two interpreter optimization techniques, replication and superinstructions. They evaluate the static and dynamic approaches to these techniques on the Gforth interpreter. In the static approach, the interpreter implementer creates replicas or superinstructions while writing the interpreter. Whereas, replicas or superinstructions are created at runtime in the dynamic variant. In 2007, Casey et al. [33] extend this work by adding experimental results from a Java VM.

In 2005, Berndl et al. [23] introduce context threading technique for virtual machine interpreters, which is based on subroutine threading. They use subroutine threaded code for regular bytecodes, dispatching each bytecode via a native call/return instruction pair. Context threading inlines the implementations of branch instructions into subroutine threaded code, giving branch prediction more context. The authors evaluate their technique for Java and OCaml interpreters on the Pentium and PowerPC architectures, and show that it significantly improves branch prediction and reduces execution time.

All of the previous work in the area of threaded code focuses on interpreter implementations using a low level systems programming language with an ahead-of-time compiler. In addition to improving array store performance—which, after all, constitutes almost half of the speedup—we present threaded code as a simple, domain-specific optimization for virtual machines having a just-in-time compiler. Furthermore, wrapping threaded code generation in annotations gives implementers a straightforward way to enable efficient interpretation, which can also be reused for several different hosted interpreters.
6.2 Targeting Virtual Machines

The idea of optimizing dynamic languages in an existing JIT environment is not new. For example, the Da Vinci Machine Project [73] extends the Java HotSpot VM to run non-Java languages efficiently on a JVM. The project introduces a new Java bytecode instruction: `invokedynamic` [87]. This instruction's intent is to improve the costly invocation semantics of dynamically typed programming languages targeting the Java virtual machine. Similar to `invokedynamic`, our system can also be adopted by JVM vendors to improve the performance of hosted language interpreters running on top of them.

In 2009, Bolz et al. [26] and Yermolovich et al. [96] describe an approach to optimize hosted interpreters on top of a VM that provides a JIT compilation infrastructure. This is particularly interesting, as this work provides a solution to circumvent the considerable implementation effort for creating a custom JIT compiler from scratch. There are two key parts in their solution. First, they rely on trace-based compilation [35] to record the behavior of the hosted program. Second, they inform the trace recorder about the instruction dispatch occurring in the hosted interpreter. Therefore, the subsequent trace compilation can remove the dispatch overhead with constant folding and create optimized machine code.

There are several differences between the approach of hierarchical layering of VMs and automatically creating threaded code at runtime. Our approach does not trace the interpreter execution, i.e., subroutine threaded code compiles a complete method, more like a conventional JIT compiler. In addition, our technique does not depend on any tracing subsystem, such as the trace compiler and recorder. Furthermore, our approach does not need to worry about bail out scenarios when the interpreter leaves a compiled trace so it does not require any deoptimization. Finally, we think that using both approaches together could have mutually beneficial results, though they inhabit opposing ends in the implementation effort/performance spectrum.
In 2012, Ishizaki et al. [56] describe an approach to optimize dynamic languages by “re-purposing” an existing JIT compiler. They reuse IBM J9 Java virtual machine and extend the JIT compiler to optimize Python. Castanos et al. [34] extend this paper to explain their repurposed “Fiorano JIT compiler” in detail. They also discuss the advantages and disadvantages of adapting an existing statically typed language JIT compiler to dynamically typed languages. Their approach and ours share the same starting point, as we both believe that developing a JIT compiler for each dynamic language from scratch is prohibitively expensive. However, their technique is still complicated to use for programming language implementers, as they need to extend the existing JIT with their own implementation language. In contrast, our approach requires only minimal mechanical transformations that are independent of the interpreted programming language (e.g., Python, JavaScript, or Ruby) to generate a highly efficient interpreter.

6.3 Profilers

Measuring execution frequencies to guide programmers to apply optimizations has a long history. In 1971, Knuth [58] reports an early study of frequency counts of each statement in Fortran programs. In 1973, Knuth and Stevenson [59] introduce an efficient algorithm for vertex (basic block) profiling.

Graham et al. [49] describe the implementation of the gprof performance analysis tool for Unix applications. They developed the tool for Berkeley Unix in 1982, and it has been part of the GNU project since 1988. The gprof tool gathers execution counts and execution time for profiled subroutines by using a combination of instrumentation and sampling. It has an augmenting compiler which instruments source code to count the number of times of each subroutine is called. The compiler inserts calls to a monitoring routine in the prologue of a profiled subroutine. The gprof tool also samples the program counter at fixed
intervals to collect execution time, so the resulting data from execution time is a statistical approximation.

Ball et al. [19, 20] introduce path profiling that records execution frequency of basic blocks or control-flow edges during an execution for purposes like performance tuning, profile-directed compilation and test coverage. They describe an algorithm to insert less instrumentation code, and place it in less frequently executed parts of the program to reduce the instrumentation overhead. Unlike gprof, their technique records exact execution frequency, not a statistical sample. In 1999, Melski et al. [67] extend this technique to collect information for interprocedural paths.

Traditional profilers perform offline, that is, they collect information in a separate run, and then use the collected information afterwards. However, dynamic compilation systems such as JIT compilers need to collect information and consume it online. Oren et al. [75] describe an online path profiling in 2002.

In 2005, Bond and McKinley [27] introduce PEP, a combination of instrumentation and sampling path and edge profiling. It uses a subset of Ball et al. [20]’s path profiling technique to identify paths with low overhead, and uses sampling to reduce the cost of storing paths.

Perf [8] is a performance analyzing tool for Linux, collecting hardware performance counters to count the number of certain hardware events, such instructions, cache misses, or branch mispredictions. It produces performance counter statistics with low overhead.

DynamoRIO [28], Pin [63], and Valgrind [71, 72] are instrumentation frameworks for building dynamic analysis tools. DynamoRIO was originally created as a dynamic binary optimization tool but has since been used for program analysis, profiling, instrumentation, optimization, and translation. It targets applications running on the Linux and Windows operating systems, and IA-32 and x86-64 instruction set architectures. Similarly, Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures. Valgrind
was originally developed as memory debugging tool for Linux, but it now includes multiple tools, such as cache and call graph profiler. *Callgrind* [91] is a profiling tool that records the function call history as a call graph. It gathers the number of instructions, numbers of calls, and calling relationship between functions.

*Javassist* [36], *Soot* [90], *ASM* [29] and *DiSL* [64] are Java bytecode manipulation and analysis frameworks. *The Java Virtual Machine Tool Interface (JVMTI)* [74] is an interface to tools that need an access to VM state, such as profiling, debugging, monitoring, and thread analysis. The interface provides support for bytecode instrumentation to modify the Java bytecode in a Java program. It can maintain exact counters or statistically sample events.

Binder et al. [24] describe their Java Profiler tool *JP* that instruments the bytecode of Java programs to profile the number of executed methods and bytecode in a given program. JP is an event-based profiler implemented in pure Java, and its implementation does not rely on JVMTI.

*Microsoft Visual Studio Profiling Tools* [69] is a single profiling environment for the languages running on the Windows .NET platform such as C, C++, Visual Basic, and C♯. It provides cross-language support which allows interaction with code written in a different programming language. It supports both instrumentation and sampling-based profiling to collect performance data.

Bergel [22] introduces *Compteur*, which is the message-based code profiler for the Pharo implementation of the Smalltalk language. Pharo is an object-oriented programming language and environment in the tradition of Smalltalk, and relies on message passing. Compteur uses message counting as a profiling metric, and shows more stable measurements than execution sampling.

Morandat et al. [70] describe *ProfileR*, a profiling tool for the VM they implement for R
language. It is an event-based profiler measuring the time spent in operations such as memory management, I/O, and foreign calls.

Our technique differs from existing profilers in several aspects: First, we instrument the AST nodes generated from the source code whereas they instrument the compiled binary code, or managed bytecode. Second, the inserted nodes in our implementation are subject to full runtime optimizations, and could be activated or deactivated at runtime. Third, some of the existing profilers use sampling, whereas our profiler is event-based tracking every occurrence of events. Fourth, some of the existing techniques require significant implementation effort, but our profiler requires little implementation. Fifth, our profiler makes it possible to compare the programs implementing the same algorithms across different languages. Lastly, our technique benefits both the language implementer and user.
Chapter 7

Conclusion

Despite its many benefits such as simplicity and flexibility, the performance of interpreters remains a primary concern for language implementers. Therefore, many language implementers design a compiler to boost the performance of their language which requires major effort investments. Our research focuses on solving the simplicity and performance dilemma for dynamic language implementations in the context of Java virtual machine. Hosted interpreters and host VM targeted compilers are two available execution choices for the languages running on a host virtual machine. This dissertation investigates the potential of combining the simplicity of the hosted interpreters with the performance of host VM targeted compilers.

We study the performance issues in hosted dynamic language interpreters running on a Java virtual machine. We analyze the bottlenecks limiting the performance of these interpreters, and describe optimizations that provide significant performance improvement. Specifically, we present two optimizations that improve interpretation performance to such a big extent that they become comparable to performance previously reserved for host VM targeted compilers. First, we optimize instruction dispatch by generating subroutine threaded code. Second, we improve array store efficiency by eliminating redundant array store type checks,
which are particularly expensive for hosted interpreters. To evaluate the impact of our system, we apply our optimizations to three hosted dynamic language implementations running on a Java virtual machine: Jython (Python implementation), Rhino (JavaScript implementation) and JRuby (Ruby implementation). Our assessment shows significant performance increase on these real-world hosted interpreters. As a result, our technique gives language implementers the opportunity to execute their language efficiently on top of a Java virtual machine without implementing a host VM targeted compiler.

Furthermore, we design a high-performance profiler framework for dynamic languages in the context of the Truffle framework that optimizes AST interpreters with a JIT compiler on a Java virtual machine. It is an event-based profiler that provides a more comprehensive profiling to further investigate dynamic language programs. Our generic profiler framework minimizes profiler implementation effort, and benefits both the language user and implementer. It makes it possible to compare languages on cross-language benchmarks implementing the same algorithms. We evaluate our profiler framework on ZipPy (Python implementation) and JRuby+Truffle (Ruby implementation), however, it is applicable to any language running on top of Truffle. Our profiler runs faster than existing profilers on average and requires modest implementation effort.
Bibliography


Appendix A

Efficient Hosted Interpreters Detailed Benchmark Results
### A.1 Jython Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Switch-based</th>
<th>Threaded Code</th>
<th>Threaded Code + Efficient Array Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>5720.50</td>
<td>3909.90</td>
<td>2450.10</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>23559.60</td>
<td>18428.90</td>
<td>11380.60</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>109690.70</td>
<td>88931.40</td>
<td>56025.30</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>57294.10</td>
<td>20835.80</td>
<td>15456.00</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>691734.60</td>
<td>250847.30</td>
<td>184617.90</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>9863.10</td>
<td>7943.80</td>
<td>6973.50</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>27931.20</td>
<td>23861.10</td>
<td>20224.80</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>63956.60</td>
<td>55659.80</td>
<td>46791.10</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>48041.40</td>
<td>18106.50</td>
<td>13207.10</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>85083.50</td>
<td>31748.40</td>
<td>23289.00</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>132694.60</td>
<td>49542.60</td>
<td>36306.80</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>74132.10</td>
<td>64983.00</td>
<td>54884.50</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>43276.10</td>
<td>18040.70</td>
<td>12070.00</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>127669.90</td>
<td>53962.60</td>
<td>35898.70</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>211816.70</td>
<td>89442.30</td>
<td>60127.80</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>42885.20</td>
<td>32065.70</td>
<td>19877.80</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>95005.40</td>
<td>71962.80</td>
<td>44428.00</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>169297.80</td>
<td>129004.30</td>
<td>77799.30</td>
</tr>
<tr>
<td>mean</td>
<td>64113.85</td>
<td>37007.32</td>
<td>26131.17</td>
</tr>
</tbody>
</table>

Table A.1: Running time of Jython interpreters in milliseconds.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>1775.60</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>7727.50</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>37053.90</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>19522.80</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>234185.20</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>7678.60</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>22518.60</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>52482.20</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>16029.60</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>28217.00</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>43812.70</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>49819.70</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>13840.20</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>40961.70</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>67965.00</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>15546.20</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>34210.80</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>61074.50</td>
</tr>
<tr>
<td>mean</td>
<td>25822.74</td>
</tr>
</tbody>
</table>

Table A.2: Running time of Jython’s host VM targeted compiler in milliseconds.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Client Compiler</th>
<th>Server Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>3708.00</td>
<td>3267.40</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>18039.80</td>
<td>9820.60</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>83300.20</td>
<td>39621.20</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>23952.20</td>
<td>10369.40</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>286496.00</td>
<td>118341.60</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>6448.40</td>
<td>6946.40</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>21387.80</td>
<td>12935.40</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>51278.80</td>
<td>24963.80</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>19118.40</td>
<td>13435.20</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>34038.20</td>
<td>22708.00</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>53092.40</td>
<td>34949.60</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>44869.40</td>
<td>22227.60</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>11853.20</td>
<td>6909.80</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>35219.80</td>
<td>19521.00</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>58596.60</td>
<td>32263.40</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>40237.40</td>
<td>16398.60</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>91309.80</td>
<td>35243.60</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>157813.40</td>
<td>63675.20</td>
</tr>
<tr>
<td>mean</td>
<td>34915.03</td>
<td>19258.74</td>
</tr>
</tbody>
</table>

Table A.3: Running time of Jython’s switch-based interpreter compiled by Hotspot’s client vs. server compiler in milliseconds.
### A.2 Rhino Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Switch-based</th>
<th>Threaded Code</th>
<th>Threaded Code + Efficient Array Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>10357.70</td>
<td>6388.80</td>
<td>5009.20</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>46359.70</td>
<td>26900.40</td>
<td>20252.60</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>213643.30</td>
<td>123634.40</td>
<td>92323.20</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>19129.30</td>
<td>7412.00</td>
<td>3161.70</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>227007.60</td>
<td>83521.40</td>
<td>33078.10</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>25149.50</td>
<td>13380.20</td>
<td>9553.70</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>97574.90</td>
<td>49477.00</td>
<td>34578.00</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>241844.10</td>
<td>121900.80</td>
<td>84698.80</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>74514.50</td>
<td>36350.50</td>
<td>23450.50</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>131059.00</td>
<td>63651.30</td>
<td>41149.50</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>204329.00</td>
<td>98717.40</td>
<td>65161.60</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>23209.20</td>
<td>10519.10</td>
<td>5582.20</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>68365.20</td>
<td>29899.40</td>
<td>15108.70</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>113024.10</td>
<td>49013.70</td>
<td>24373.70</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>46793.70</td>
<td>19618.20</td>
<td>10634.60</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>104238.30</td>
<td>42351.70</td>
<td>22930.10</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>183986.40</td>
<td>74243.30</td>
<td>39084.20</td>
</tr>
<tr>
<td>mean</td>
<td>75539.30</td>
<td>35542.81</td>
<td>21144.80</td>
</tr>
</tbody>
</table>

Table A.4: Running time of Rhino interpreters in milliseconds.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>2488.10</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>9228.50</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>39875.90</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>2801.60</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>29081.70</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>7249.80</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>25800.80</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>63300.00</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>30041.50</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>52604.60</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>79158.30</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>-</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>3768.10</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>9980.90</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>15969.00</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>6015.30</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>12950.40</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>21790.90</td>
</tr>
<tr>
<td>mean</td>
<td>15118.66</td>
</tr>
</tbody>
</table>

Table A.5: Running time of Rhino’s host VM targeted compiler in milliseconds.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Client Compiler</th>
<th>Server Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>5810.80</td>
<td>5597.40</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>26458.80</td>
<td>21195.20</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>124295.80</td>
<td>94393.60</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>6798.40</td>
<td>7863.20</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>79603.80</td>
<td>85274.40</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>12510.80</td>
<td>13062.20</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>49069.00</td>
<td>42501.60</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>122127.40</td>
<td>87273.80</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>44954.60</td>
<td>32361.80</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>79785.40</td>
<td>56825.60</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>126413.40</td>
<td>87924.20</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>10212.40</td>
<td>11058.20</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>30147.80</td>
<td>29362.00</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>50013.40</td>
<td>47627.60</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>18207.80</td>
<td>20450.00</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>41200.60</td>
<td>44009.20</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>75697.80</td>
<td>77622.40</td>
</tr>
<tr>
<td>mean</td>
<td>36149.37</td>
<td>32948.15</td>
</tr>
</tbody>
</table>

Table A.6: Running time of Rhino’s switch-based interpreter compiled by Hotspot’s client vs. server compiler in milliseconds.
Table A.7: Running time of JRuby interpreters in milliseconds.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>896.00</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>2765.33</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>13730.00</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>4594.00</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>51611.33</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>7993.00</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>29111.67</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>71429.00</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>11708.33</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>20632.33</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>31942.33</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>14507.00</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>3978.67</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>10764.67</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>17659.33</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>3731.67</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>8714.33</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>15409.67</td>
</tr>
<tr>
<td>mean</td>
<td>11013.91</td>
</tr>
</tbody>
</table>

Table A.8: Running time of JRuby’s host VM targeted compiler in milliseconds.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Client Compiler</th>
<th>Server Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 12</td>
<td>2554.20</td>
<td>3085.40</td>
</tr>
<tr>
<td>binarytrees 14</td>
<td>12107.00</td>
<td>10166.60</td>
</tr>
<tr>
<td>binarytrees 16</td>
<td>57581.20</td>
<td>40341.20</td>
</tr>
<tr>
<td>fannkuchredux 9</td>
<td>5594.20</td>
<td>6066.20</td>
</tr>
<tr>
<td>fannkuchredux 10</td>
<td>69439.40</td>
<td>71013.40</td>
</tr>
<tr>
<td>fasta 250000</td>
<td>10240.00</td>
<td>9147.60</td>
</tr>
<tr>
<td>fasta 1000000</td>
<td>39658.00</td>
<td>26762.20</td>
</tr>
<tr>
<td>fasta 2500000</td>
<td>99651.20</td>
<td>61782.20</td>
</tr>
<tr>
<td>mandelbrot 600</td>
<td>12728.60</td>
<td>14020.00</td>
</tr>
<tr>
<td>mandelbrot 800</td>
<td>22465.00</td>
<td>23599.40</td>
</tr>
<tr>
<td>mandelbrot 1000</td>
<td>35086.00</td>
<td>36006.80</td>
</tr>
<tr>
<td>meteor 2098</td>
<td>25657.60</td>
<td>21521.80</td>
</tr>
<tr>
<td>nbody 100000</td>
<td>6403.60</td>
<td>6773.00</td>
</tr>
<tr>
<td>nbody 300000</td>
<td>19035.60</td>
<td>18612.00</td>
</tr>
<tr>
<td>nbody 500000</td>
<td>31868.20</td>
<td>30483.60</td>
</tr>
<tr>
<td>spectralnorm 500</td>
<td>7366.20</td>
<td>7600.00</td>
</tr>
<tr>
<td>spectralnorm 750</td>
<td>16336.60</td>
<td>14497.00</td>
</tr>
<tr>
<td>spectralnorm 1000</td>
<td>27699.20</td>
<td>23580.40</td>
</tr>
<tr>
<td>mean</td>
<td>18884.47</td>
<td>17387.75</td>
</tr>
</tbody>
</table>

Table A.9: Running time of JRuby’s switch-based interpreter compiled by Hotspot’s client vs. server compiler in milliseconds.
Appendix B

Efficient Profiler Framework Detailed Benchmark Results
B.1 Python Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ZipPy</th>
<th>PyPy</th>
<th>CPython</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 16</td>
<td>9.840</td>
<td>22.441</td>
<td>82.540</td>
</tr>
<tr>
<td>fannkuchredux 11</td>
<td>5.750</td>
<td>9.818</td>
<td>442.804</td>
</tr>
<tr>
<td>mandelbrot 4000</td>
<td>10.658</td>
<td>14.641</td>
<td>119.160</td>
</tr>
<tr>
<td>nbody 5000000</td>
<td>18.427</td>
<td>11.714</td>
<td>111.003</td>
</tr>
<tr>
<td>pidigits 15000</td>
<td>17.406</td>
<td>13.036</td>
<td>13.683</td>
</tr>
<tr>
<td>richards 200</td>
<td>1.171</td>
<td>1.549</td>
<td>55.987</td>
</tr>
<tr>
<td>spectralnorm 3000</td>
<td>3.769</td>
<td>3.773</td>
<td>320.555</td>
</tr>
</tbody>
</table>

Table B.1: Running time of Python implementations in seconds.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ZipPy+Profiler</th>
<th>PyPy+Profiler</th>
<th>CPython+Profiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 16</td>
<td>26.557</td>
<td>51.127</td>
<td>132.149</td>
</tr>
<tr>
<td>fannkuchredux 11</td>
<td>5.929</td>
<td>9.714</td>
<td>443.217</td>
</tr>
<tr>
<td>mandelbrot 4000</td>
<td>42.557</td>
<td>45.420</td>
<td>194.518</td>
</tr>
<tr>
<td>nbody 5000000</td>
<td>18.438</td>
<td>11.066</td>
<td>112.487</td>
</tr>
<tr>
<td>pidigits 15000</td>
<td>17.391</td>
<td>13.038</td>
<td>13.250</td>
</tr>
<tr>
<td>richards 200</td>
<td>8.002</td>
<td>8.691</td>
<td>91.923</td>
</tr>
<tr>
<td>spectralnorm 3000</td>
<td>28.858</td>
<td>19.887</td>
<td>415.789</td>
</tr>
</tbody>
</table>

Table B.2: Running time of Python method profilers in seconds.
### B.2 Ruby Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JRuby+Truffle</th>
<th>JRuby</th>
<th>MRI (CRuby)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 16</td>
<td>22.81</td>
<td>57.36</td>
<td>44.41</td>
</tr>
<tr>
<td>fannkuchredux 11</td>
<td>9.48</td>
<td>88.39</td>
<td>277.92</td>
</tr>
<tr>
<td>mandelbrot 4000</td>
<td>3.35</td>
<td>368.95</td>
<td>288.94</td>
</tr>
<tr>
<td>nboby 5000000</td>
<td>2.87</td>
<td>50.65</td>
<td>157.97</td>
</tr>
<tr>
<td>pidigits 15000</td>
<td>16.51</td>
<td>18.34</td>
<td>17.20</td>
</tr>
<tr>
<td>richards 200</td>
<td>1.47</td>
<td>17.05</td>
<td>23.74</td>
</tr>
<tr>
<td>spectralnorm 3000</td>
<td>5.90</td>
<td>383.48</td>
<td>190.56</td>
</tr>
</tbody>
</table>

Table B.3: Running time of Ruby implementations in seconds.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JRuby+Truffle+Profiler</th>
<th>JRuby+Profiler</th>
<th>MRI (CRuby)+Profiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees 16</td>
<td>40.98</td>
<td>122.11</td>
<td>1094.53</td>
</tr>
<tr>
<td>fannkuchredux 11</td>
<td>9.48</td>
<td>96.31</td>
<td>7738.51</td>
</tr>
<tr>
<td>mandelbrot 4000</td>
<td>3.36</td>
<td>373.69</td>
<td>6756.21</td>
</tr>
<tr>
<td>nboby 5000000</td>
<td>4.67</td>
<td>157.95</td>
<td>2466.92</td>
</tr>
<tr>
<td>pidigits 15000</td>
<td>16.79</td>
<td>18.86</td>
<td>17.21</td>
</tr>
<tr>
<td>richards 200</td>
<td>7.06</td>
<td>56.00</td>
<td>394.17</td>
</tr>
<tr>
<td>spectralnorm 3000</td>
<td>30.81</td>
<td>535.24</td>
<td>5068.78</td>
</tr>
</tbody>
</table>

Table B.4: Running time of Ruby method profilers in seconds.