TOKYO INSTITUTE OF TECHNOLOGY

DOCTORAL THESIS

Supporting multi-scope and multi-level compilation in a meta-tracing just-in-time compiler

---

Author:
Yusuke IZAWA

Student Number:
20D30016

Supervisor:
Prof. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

March 1, 2023

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Doctor of Science

Supporting multi-scope and multi-level compilation in a meta-tracing just-in-time compiler

by Yusuke IZAWA

Just-in-Time (JIT) compilation technologies are widely used in today's virtual machines (VMs) to achieve high performance. There are several ways to compile in terms of compilation scopes. The most common is to compile on a per-method basis, while others compile on an execution-history basis or on an arbitrary program region. In addition, today's virtual machines use multiple levels of compilation for different parts of the code: for example, optimizing heavyweight compilation is used for frequently executed parts of the code, and lightweight compilation is used for code parts executed moderately often.

Many of today's programming languages are realized using virtual machines, and among them, the meta-JIT compiler framework, which generates a virtual machine for a language from an interpreter that defines the language specification, is attracting attention. Meta-JIT compiler frameworks, such as RPython and Truffle/Graal, are used in the field of a programming language implementation, and the usefulness of the system was demonstrated by actually realizing VMs of Python, Ruby, R, JavaScript, and so forth.

However, the current JIT compilers provided by meta-JIT compiler frameworks have only a fixed compilation policy and cannot compete with various compilation policies developed by JIT compilers in VMs for specific languages.

To further improve the performance of the JIT compiler generated by a meta-JIT compiler framework, a new compilation technology that can support multiple compilation scopes and levels would be beneficial. The present dissertation shows that an interpreter, which is a language specification given to the meta-JIT compiler framework, is not only the semantic definition of the language but also the compilation policy and that multiple compilation policies can actually be realized on the meta-tracing JIT compiler framework.

The present dissertation proposes multi-role meta-tracing JIT compilation, that mixes different compilation scopes and levels in a meta-tracing JIT compiler framework. This technique is achieved by using a hint instruction, which is a pseudo-function inserted into an interpreter.

An implementation of a prototype framework called BacCaml and evaluation with it show that there exists a program that runs faster with multi-scope compilation combining method and trace-based compilations than with a single compilation scope. In addition, the implementation and evaluation using RPython validate that supporting multi-scope compilation in RPython, where method-based lightweight compilation is added to RPython, which originally had only a heavyweight trace-based compiler, improves the overall performance of RPython in a real-world workload-simulated application. The results of the experiment conducted in a real-world workload-simulated show that the two-level compilation mixing the method-based lightweight and trace-based heavyweight compilation can improve the performance against the single-level tracing compilation in about 14%.

Acknowledgements

Words cannot express my gratitude to my supervisor Prof. Hidehiko Masuhara and my advisor Dr. Carl Friedrich Bolz-Tereick for their invaluable patience and feedback. I also would like to thank thank Prof. Youyou Cong for her kind support on my study. I also could not have undertaken this journey without my defense committee, who generously provided knowledge and expertise. Additionally, this endeavor would not have been possible without the generous support from the Japan Society for the Promotion of Science and Japan Society and Technology Agency, who financed my research.

I am grateful to my lab mates for their editing help, late-night feedback sessions, and moral support. My appreciation also goes out to my family and friends for their encouragement and support all through my studies.

Contents

1 Introduction 1

1.1 Research Context 2
1.2 Problem Statement 3
1.3 Research Goals and Outlines 5

2 Background 9

2.1 JIT Compilation 9
2.1.1 JIT Compilation Strategies 10
2.1.2 Method-Based JIT Compilation 11
2.1.3 Trace-Based JIT Compilation 12
2.2 Meta-JIT Compiler Framework 13
2.2.1 Self-Optimizing Interpreter 14
Interpreter Definition in a Self-Optimizing Interpreter . 14
Meta-Compilation in Self-Optimizing Interpreter . . . 15
Hint Instructions in Truffle 18
Truffle Host Inlining 19
2.2.2 Meta-Tracing JIT Compiler 20
Interpreter Definition in a Meta-Tracing JIT Compiler . 21
Meta-Compilation in Meta-Tracing JIT Compiler 22
Hint Instructions in RPython 22

3 Motivation and Proposal 25

3.1 Scope of Compiling a Source Program 25
3.2 Level of Compiling a Source Program 27
3.3 Dilemma: Hard to Extend Generated VMs from a Meta-JIT Compiler Framework 29
3.4 Proposal: Multi-Role Meta-Tracing JIT Compilation 30
3.4.1 System Overview: Multi-Role Meta-Tracing JIT Compiler Framework 32
3.4.2 Underlying Techniques for Multi-Scope Compilation . 33
Hint Instructions and Compilation Overview 34
3.4.3 Underlying Techniques for Multi-Level Compilation . 35

4 BacCaml – a Proof-of-Concept Multi-Scope Meta-Tracing JIT Compiler 39

4.1 Introduction 39
4.2 Mixing The Two Compilation Strategies in Meta-Level 41
4.2.1 Method-Based Compilation by Tracing 41
4.3 Stack Hybridization 46

4.3.1 Combination Problem 46
4.3.2 Bridging Native Code with Different Calling Conventions 48
4.4 Evaluation 49
4.4.1 Setup 50
Methodology 50
Threats to Validity 51
4.4.2 Standalone JIT Microbenchmark 51
4.4.3 Multi-Scope JIT Experiment 53
Methodology 53
Results of Multi-Scope JIT Experiment 54
4.5 Related Work 55
4.6 Conclusion 57

5 Threaded Code Gen. with a Real-World Meta-Tracing .. 59
5.1 Introduction 60
5.2 Threaded Code Generation 60
5.2.1 Threaded Code 61
5.2.2 The Compilation Principle 61
5.2.3 Method-traversal Interpreter 64
Conditional branch 65
Back-edge instruction 66
Function call 66
Function return 67
5.2.4 Trace Stitching 67
Guard Patching in Trace Stitching 68
5.2.5 Shallow Tracing 69
5.3 Runtime Techniques for Multilevel Compilation 70
5.3.1 Implementaiton Details 71
5.4 Optimization for Threaded Code Generation with Interpreter in the Meta-Tracing JIT Compiler 72
5.4.1 Inline Caching in Method-Traversal Interpreter 72
5.5 Preliminary Evaluation Using Simulated Threaded Code Generation 74
5.5.1 Simulated Threaded Code Generation (STCG) in PyPy 75
5.5.2 Setup 76
System 76
Implementation 76
Programs for Experiments 76
Methodology 76
5.5.3 Results of Experiment 1: The Overhead of Our STCG . 77
5.5.4 Results of Experiment 2: The Stable Speed 78
5.5.5 Discussion 78
5.6 Evaluation and Experiments in PySOM and Multilevel RPython 79
5.6.1 Microbenchmark Evaluation 79
Setup 80
Methodology 80

Code Sizes and Compilation Times 81
Peak Performance at Steady State 82
5.6.2 Multilevel JIT Experiment 83
Setup 83
Methodology 83
Results of the Experiment and Discussion 84
5.7 Related work 85
5.7.1 Improving an Interpreter's Performance 85
5.7.2 Template JIT Compilation 85
5.7.3 Ahead-of-Time Compilation 86
5.7.4 Introducing a New Behavior into a Meta-JIT Compiler 86
5.8 Conclusion 86

6 Conclusion **99**

Bibliography **101**

List of Figures

2.1 An overview of how a JIT compiler works in a VM. 10
2.2 The difference between method- and trace-based JIT compilation strategies. The linked control flow and blocks represent a method. Areas painted or lines drawn represent a compilation target. 12
2.3 A simple Python program and its' recorded trace. 13
2.4 System structure of a meta-JIT compiler framework. 13
2.5 Definition of an AST node base class, which is retrieved from [Würthinger et al., 2012] 15
2.6 Implementation of a `while` block, which is retrieved from [Würthinger et al., 2012]. 16
2.7 A transition between different types in an AST tree writing, which is retrieved from [Würthinger et al., 2012]. 16
2.8 An example of function `add` in a dynamic language. 16
2.9 Type-specialized execute methods, which are cited from [Würthinger et al., 2012]. 17
2.10 Integer add node using return type specialization as cited in [Würthinger et al., 2012]. 17
2.11 Implementation of a `while` block using the Truffle DSL 18
2.12 A binary node that is annotated by `NodeChild`. 18
2.13 A node representing `add`. 18
2.14 Example of a bytecode interpreter written in Truffle. 20
2.15 VM generation and JIT compilation flow in RPython. Solid and dashed lines represent data and control flow, respectively. 21
2.16 An example interpreter definition written in RPython. 22

3.1 Simplified overview of a managed VM. 28
3.2 Simplified overview of a generated VM from a meta-JIT compiler framework. 29
3.3 Overview of a multi-role meta-tracing JIT compiler framework. 31
3.4 Two approaches to a specify JIT compilation policy in BacCaml and Multilevel RPython. 33
3.5 Compilation Overview of Method JIT by Tracing. 36
3.6 Compilation overview of lightweight compilation in Multilevel RPython. 36

4.1 The overviews of a multi-scope meta-tracing JIT compiler framework. 40

4.2 Examples how our method-based compilation works. Each left-hand side is the control-flow of a target source program that represents one method, and each right-hand side is a result. “entry” and “return” means the entry point and exit poitn of a target method, respectively. 44
4.3 Interpreter definition styles. For managing a return address/value, left-hand side style uses a host-language’s (system provided) stack, but right-hand side uses a developer-prepared stack data structurpe. 46
4.4 Example of Combination Problem. Gray background code is compiled by method JIT, and blue lined code is compiled by tracing JIT. 47
4.5 A sketch of a interpreter definition with Stack Hybridization. Some hint functions (e.g., `can_enter_jit` and `jit_merge_point`), and other definitions are omitted for simplicity. `US` and `HS` represents user-stack and host-stack, respectively. 49
4.6 Results of standalone JIT microbenchmarking. The five programs on the left have a complex control flow, and the remaining programs have a straight control flow. The error bars represent the standard deviations. 52
4.7 Target programs written in MinCaml−− used for the multi-scope JIT experiment. 54
4.8 Results of multi-scope JIT microbenchmarking. X-axis represents the name of a target program, and Y-axis represents speedup ratio relative to the interpreter-only execution. Higher is better. The error bars represent the standard deviations. 55

5.1 Overview of Multilevel RPython: there exists two different interpreters in the generated VM. At VM generation time, Multilevel RPython generates a VM from an interpreter instrumented with hint instructions used for multilevel compilation. At runtime, the different two interpreters are used for lightweight and heavyweight compilations. First, a source program is run on the interpreter for lightweight compilation. When hot spots are detected, the execution is switched to the interpreter for heavyweight compilation from one for lightweight compilation. This transition is performed by the interpreter shifting technique described in Section 5.3. . 87
5.2 An overview of how threaded code works. 88
5.3 A sketch of how RPython method-based lightweight JIT compiler works. From the target function in the left-hand side, it generates the trace tree shown in the right-hand side. . 88
5.4 Tracing the entire of a function with method-traversal interpreter. 88
5.5 The working flow of trace stitching. 89

5.6 Overview of trace-stitching. This shows how we resolve the relations between guard failures and bridges. 89
5.7 Overview of the runtime techniques for multilevel compilation. The interpreter shifting defines the strategy for transitioning between compilation levels. Note that each interpreter shown on the right-hand side corresponds to each compilation level. For each interpreter, the drivers profile the execution and starts JIT compilation when the threshold is exceeded. 90
5.8 Calling a compiled method w/o inline caching. 90
5.9 Overview of inline caching in threaded code generation. . . . 91
5.10 The results of the size of traces to compile and compilation time including tracing. In all results the Y-axis means PyPy 3.7 with our simulated threaded code generation (STCG)'s relative value to PyPy 3.7–7.3.5's tracing JIT compiler. The X-axis stands for the name of every program. The left-hand side shows the relative trace size, and the right-hand size is the relative compilation time. Lower is better. 92
5.11 The results of a preliminary benchmark experiment. In all results the Y-axis means speed up ratio of the threaded code generation compared with the interpreter-only execution, and the X-axis stands for the name of every program. The error bars mean standard deviations. Higher is better. 93
5.12 Speed-up ratio of STCG (left-hand side) and PyPy's tracing JIT compiler (right-hand side) related to the interpreter. They are executed on PyPy's original micro benchmark suite plus our original ones. X-axis and Y-axis mean every iteration and speed-up ratio standardized to interpreter execution, respectively. Dots are plotted every five iterations. 94
5.13 Compilation times and code sizes on micro-benchmark programs. The x-axis and y-axis mean a bytecode size (byte) and consumed compilation time (ms), respectively. 95
5.14 Peak performance of threaded code generation, tracing JIT and interpreter execution. Relative elapsed times normalized to interpreter execution. The blue and red lines mean threaded code generation and tracing JIT, respectively. Lower is better. . 95
5.15 Number of method invocations in the simulated real-world workload application. 96
5.16 Visualization of which methods compile at which level by defined thresholds using Figure 5.15. 96
5.17 Results of the experiment in the simulated real-world workload application. 97

List of Tables

2.1 A survery on JIT compilation strategies and their compilation units and implementations. 11

5.1 Benchmark descriptions. 81
5.2 Micro benchmark programs and their bytecode sizes. Standard libraries of the PySOM system is excluded. 82
5.3 The threshold for the multilevel JIT experiment. 84

List of Listings

1 Brief example of an interpreter instrumented with the hint instructions for Method JIT by Tracing. 35
2 Brief example of interpreter definition for lightweight compilation. 37
3 Skeleton of method-traversal interpreter and subroutines decorated with `dont_look_inside`. 63
4 An example bytecode with the control flow shown in Figure 5.4. 64
5 An example program corresponding to Listing 4 64
6 Definition of JUMP_IF. 65
7 Definition of JUMP. 65
8 Definition of RET. 66
9 The trace generated temporarily from a method-transverse interpreter. 67
10 Stitched traces. One linear trace is converted into one trace and one bridge, and are connected to a guard failure. 68
11 Overview of how the decorator `enable_shallow_tracing` works. The left-hand side is an interpreter that a language developer writes. The right-hand side is the actual executable interpreter after expanding `enable_shallow_tracing`. 70
12 Before and after applying shallow tracing to the function `f` are shown on the right-hand side of the Listing 11. During shallow tracing, all flags are activated (left side), but they are finally deactivated in the resulting trace (right side). 71
13 Overview of interpreters for lightweight and heavyweight compilation. 72
14 Overview of the interpreter shifting mechanism. The left-hand and right-hand sides show the definitions of JIT policy-shifting exceptions and the interpreter shifting loop, respectively. 73
15 Interpreter definition of a dispatch loop and a handler for CALL. 74
16 Produced trace at tracing CALL instruction with the interpreter shown in Listing 15. 74
17 Interpreter definition, which is based on Listing 15, instrumented to enable inline caching. 75
18 Produced traces at tracing CALL instruction with virtual instructions to enable inline caching. 75

Chapter 1

Introduction

Contents

1.1 Research Context 2

1.2 Problem Statement 3

1.3 Research Goals and Outlines 5

With our lives becoming diversified, various devices have appeared: not only traditional on-premise servers and workstations, but also personal computers, smartphones, and laptops. Additionally, software applications such as social networks, video streaming services, and teleconference tools have emerged and have been enriching our lives. In response to these new technologies, a variety of programming languages have been developed to accommodate a variety of use cases and software productions. In particular, virtual machines (VMs) such as Java, C♯, Ruby, Python, and PHP have secured their position in today's software development environment.

Given the diversity of devices and applications, there exists an urgent need to process a program efficiently in a managed language runtime. Regarding this request, most VMs employ just-in-time (JIT) compilation. This technique has had a huge impact on improving runtime performance, and many VMs such as Java virtual machine (JVM), Common Language Runtime (CLR), and JavaScript engines equipped with a JIT compiler.

Alongside a JIT compilation scope, many optimization techniques in a VM, including method/function inlining, data flow analysis, loop unrolling and peeling, escape analysis, memory allocation optimization, and so forth have also been studied [Arnold et al., 2000; Budimlic and Kennedy, 1997; Choi et al., 1999a; Ishizaki et al., 1999]. Since today's VMs must run a huge variety of programs such as batch processing applications, data handling scripts, and server-side applications, VMs need to balance code quality with compilation time.

There are many techniques for JIT compilers, which can be classified by several criteria. A compilation scope is one of such criterion. There is a wide range of JIT compilation policies, and they can be classified in terms of the scope of compilation. The most common is method-based compilation, which is used in several JVMs, including HotSpot™ and J9, and Android

Runtime (ART). For dynamic languages, trace-based compilation, which is based on the actual executed path of a program, is used to achieve their high performance. In addition, region- or basic block-based compilation policies are also used in today's VMs such as Ruby and PHP.

A compilation level is another one of the criterion for the classification. VM developers have designed the compilation levels and classified them into various execution stages. Short-lived applications, like batch processing or data handling scripts, are applied only to lightweight compilation, which generates machine code quickly. On the other hand, for a long-lived application like a server-side application, these applications are applied to heavyweight compilation, generating fully optimized code to achieve better peak performance at a steady state.

At the same time, language developers have struggled to build high-performance VMs efficiently because developing an elaborate and sophisticated VM comes with the trade-offs of the simplicity, clarity, and comprehensiveness of its implementation. A meta-JIT compiler framework reduces these trade-offs. Not only their ability to reduce the engineering effort for VM development, but also their ability to generate a high-performance VM powered by a dedicated JIT compiler. A wide range of VMs have been implemented with the framework, such as for Ruby, Python, PHP, Smalltalk, and so forth, exhibiting the same or higher performance as their already existing VMs.

1.1 Research Context

Just-In-Time Compilation For software users, one of the most important things is that applications run fast. Basically, the common execution was performed by interpreting a bytecode program, which is a form of an intermediate representation where on instruction corresponds to one byte, in VMs. A bytecode program is obtained as a result of parsing the source program. Because this interpretation was slower than ahead-of-time (AOT) compilation, many researchers and developers have studied another approach to run a bytecode program faster than interpretation. One of the successful studies is just-in-time (JIT) compilation [Aycock, 2003]. JIT compilation is a technique that is used to translate a program into machine code at runtime. This technique was initially developed on the Smalltalk-80 [Deutsch and Schiffman, 1984] system, and successors such as the initial version of the Java VM [Venners, 1998]. In particular, a hot pot JIT compiler converts frequently executed program area, known as a *hot spot*, into machine code at runtime. This technique is widely used in current Java VMs [Paleczny, Vick, and Click, 2001] and JavaScript [Gal et al., 2009] VMs because of its good performance impact at runtime.

Meta-JIT Compiler Framework Traditionally, the developers of VMs basically write an interpreter, JIT compiler, garbage collector (GC), and so forth from scratch. In contrast, developers of VMs with a meta-JIT compiler framework only write an interpreter because a meta-JIT compiler framework can generate a high-performance VM from the given interpreter definition. Currently, there are RPython [Bolz et al., 2009] and Truffle/Graal [Würthinger et al., 2012] frameworks. RPython was originally developed for PyPy [Rigo and Pedroni, 2006], which is an alternative fast Python implementation. Truffle/Graal is a part of the GraalVM [Würthinger et al., 2017] project, which was developed by Oracle Labs. Because of their usefulness in language implementation, many languages such as Python, Ruby, R, and JavaScript have been implemented with the framework and have proved their high performance and effectiveness.

1.2 Problem Statement

There is a wide range of JIT compilation policies in existing VMs. These can be divided into four policies according to the scope of the compilation. The most standard compilation scope is the method-based one [Deutsch and Schiffman, 1984; Paleczny, Vick, and Click, 2001; Ungar and Smith, 1987], where the compilation scope is a method (or function). Furthermore, trace-based compilation [Bala, Duesterwald, and Banerjia, 2000; Gal et al., 2009; Rigo and Pedroni, 2006], which compiles the execution path of a program, is also being actively studied. In addition to these two compilation policies, there exist a variety of policies such as region-based compilation [Hank, Hwu, and Rau, 1995; Ottoni, 2018; Suganuma, Yasue, and Nakatani, 2006], which is based on an arbitrary region in a program, or basic block compilation [Bellard, 2005; Chevalier-Boisvert and Feeley, 2015, 2016], which is based on the basic blocks in a program.

First, each policy has its own advantages and disadvantages. Method-based compilation is a standard policy used in many of today's VMs and has matured, but it is needed to manage compilation code size because method-based compilation sometimes covers infrequently-executed ares of code. Trace-based compilation can generate aggressively optimized code and can avoid compiling less executed code, which is called dead code. However, when trace-based compilation traces a program with a complex control flow, it can fail into the path-divergence problem [Huang, Masuhara, and Aotani, 2016; Inoue et al., 2011; Izawa and Masuhara, 2020], where there is a costly fallback to an interpreted execution. In region-based compilation, which is built in HHVM, a VM for PHP and Hack languages[1], can flexibly select the scope of compilation and is potentially a compromise between method- and trace-based compilations. This type of compilation is promising, because it can take advantage of method- and trace-based compilations. However, the current region-based compiler is only targeted for PHP and Hack languages, so compilers need to be rebuilt from scratch when applying to other languages.

---

[1] https://hhvm.com/

Second, today's VMs must run numerous kinds of applications. VMs not only run short-lived batch processing applications and data processing scripts but also long-lived sever-side applications. To cope with this situation and achieve optimal performance anytime, today's VMs support multiple levels of optimization in their VMs. They use different interpreters or compilers, depending on their runtime situations. For example, short-lived scripts are executed on an interpreter or compiled by a lightweight JIT compiler, and long-lived applications are compiled using a heavyweight JIT compiler. In today's VMs, HotSpot uses the C1/C2 compilers [Kotzmann et al., 2008; Paleczny, Vick, and Click, 2001]. The C1 compiler [Kotzmann et al., 2008] compiles a method and emits machine code quickly instead of applying optimizations aggressively. On the other hand, the C2 compiler [Paleczny, Vick, and Click, 2001] generates highly optimized machine code, but requires more compilation time. In addition, V8 has Sparkplug/Turbofan compilers[2] and JavaScriptCore has Baseline/DFG/FTL compilers[3].

Unlike language-dependent managed language runtimes, the features described above are not easily realized in the generated VMs from a meta-JIT compiler framework because generated VMs basically use the fixed components provided by meta-JIT compiler frameworks, so VM components that include a JIT compiler and a GC are limited regarding how they can be extended. For example, a generated JIT compiler from RPython basically uses trace-based compilation, and a generated JIT compiler from Truffle/Graal uses method-based compilation. In addition, in terms of an optimization level, generated JIT compilers only support heavyweight compilation, which is a way to generate fast machine code by applying many optimization techniques. In addition, there are limited ways to change their compilation levels. Currently, it is difficult to add a new compilation level without modifying the meta-JIT compilers themselves.

Several works in RPython [Bauman et al., 2015; Bolz et al., 2011b; Huang, Masuhara, and Aotani, 2016] have found that taming an interpreter definition that are needed to perform meta-tracing JIT compilation can influence how RPython's meta-tracing JIT compiler works. To be more specific, the placement of a hint instruction called `can_enter_jit` in the interpreter written in RPython changed the behavior of tracing and improved the performance of the meta-tracing JIT compiler. In other words, an interpreter can be seen as a specification of a compiler in the context of meta-tracing JIT compilation. The present dissertation shows how those approaches can be extended to achieve a multi-scope and multilevel compilation system without drastically modifying the existing RPython meta-JIT compiler.

To conclude, among today's VMs, there exist many JIT compilation scopes, each with their own advantages. Therefore, a compilation scope based on a program region that can compile the scope based on a trace, method, and a mixture of both is promising. In addition, today's VMs support multiple optimization levels by creating and managing several compilers to balance the code quality and compilation time. However, existing meta-JIT compilers support only a single JIT compilation scope and a few levels (only an interpreter and a heavyweight JIT compiler) of execution because generated VMs are not able to be modified to support multiple compilation scopes or multiple optimization levels. However, several previous works found that there was a way to influence the behavior of a meta-tracing JIT compiler by taming an interpreter definition with hint instructions. The ability of virtual instructions can be useful in adding new compilation scopes and optimization levels in a meta-tracing JIT compiler framework. In summary, the two problems to address are the following:

---

[2] `https://v8.dev/blog/sparkplug`

[3] `https://webkit.org/blog/10308/speculation-in-javascriptcore/`

Meta-JIT compiler frameworks do not support multi-scope compilation. Today's VMs have a variety of JIT compilation policies, and they have their own advantages and disadvantages. To take advantage of each compilation policy, a region-based compilation that can compile the scope of a trace, method, and mixture of the two is considered promising. However, meta-JIT compilers currently support only a single compilation scope. To achieve high performance, achieving multi-scope compilation in a meta-JIT compiler framework is needed.

Few studies on a multilevel compilation in meta-JIT compiler frameworks. The current meta-JIT compilers always perform heavyweight compilations. This policy is not always useful for realistic usage in a VM; some programs run longer, but others run shorter. Full optimization is not always useful, so the ability to shift or balance the level of optimization in a meta-JIT compiler framework is needed.

1.3 Research Goals and Outlines

The main research aim of the present dissertation is as follows:

> *A new compilation scope and optimization level in a meta-tracing JIT compiler framework using hint instructions in an interpreter.*

In the current dissertation, the author presents an abstraction and implementation of a *multi-role compilation in a meta-tracing JIT compiler framework* that can change its compilation scopes and levels. The author shows that a multi-scope meta-tracing JIT compilation is possible by implementing a proof-of-concept framework called BacCaml. After implementing BacCaml, the author presents an approach to enable multilevel JIT compilation and optimization in a real-world meta-tracing JIT compiler framework: RPython.

The present dissertation is structured as follows:

Chapter 2 outlines the existing JIT compilation and meta-JIT compilation techniques used in today's VM. Although there are many strategies for JIT and meta-JIT compilations, the fundamental concept is simple and clear. Furthermore, it presents a meta-JIT compiler framework, which is a fundamental technique for multi-role meta-tracing JIT compilation.

Chapter 3 states the motivation for the multi-role meta-tracing JIT compiler framework. The technique has two foundations: the scope of the compiled code and optimization level of the code. First, this chapter states the motivation behind the mixing of different compilation scopes and why a mixture of different compilation level is needed. Then, the chapter gives the abstractions for the two foundations.

Chapter 4 shows a proof-of-concept multi-scope meta-tracing JIT compiler framework: BacCaml. Based on the examinations denoted in the previous chapter, first, this chapter demonstrates how multi-scope meta-tracing JIT compilation is achieved by using the two main JIT compilation policies: trace- and method-based compilation strategies. Second, the chapter introduces BacCaml, a multi-scope meta-tracing JIT compiler framework that amalgamates trace- and method-based compilations into a meta-compilation. Third, the chapter validates a new multi-scope compilation approach, showing that it can actually work. It evaluates the performance of BacCaml implementation and observes the behavior and performance of each compilation policy in the multi-scope meta-tracing JIT compiler framework in running microbenchmark programs. Furthermore, it performs an experiment investigating whether there exists a program that can speed up by the multi-scope JIT compilation.

Chapter 5 illustrates the application to the real-world meta-tracing JIT compiler framework: RPython. Based on the abstractions denoted in Chapter 3, this chapter first introduces the concept and approach to threaded code generation with RPython. In threaded code generation, two compilations that are different regarding the scope of the compilation and optimization level coexist in a meta-JIT compiler framework. Second, the chapter introduces an implementation of threaded code generation with RPython. For a demonstration of how threaded code generation works in a real-world language, it uses PySOM, which is a subset of Smalltalk implementation on top of RPython. This chapter explains several implementation techniques for threaded code generation in RPython. It also discusses the potential performance in Python by simulating the behavior of threaded code generation using the PyPy interpreter. In addition, the chapter shows a preliminary evaluation result to evaluate the performance of threaded code generation in a small language written in RPython before the evaluating the performance of PySOM with threaded code generation and discussing the performance characteristics of threaded code generation compared with interpreter and tracing JIT executions. Finally, the performance of multi-level compilation that mixes threaded code generation and tracing JIT compilation is evaluated in an application with a real-world workload. A simulated application synthetically combining benchmark programs is prepared for this experiment.

In the simulated application, the performance of multilevel compilation is evaluated against tracing JIT and interpreter executions.

Chapter 6 concludes the dissertation.

Chapter 2

Background

Contents

2.1 JIT Compilation 9
  2.1.1 JIT Compilation Strategies 10
  2.1.2 Method-Based JIT Compilation 11
  2.1.3 Trace-Based JIT Compilation 12
2.2 Meta-JIT Compiler Framework 13
  2.2.1 Self-Optimizing Interpreter 14
  2.2.2 Meta-Tracing JIT Compiler 20

This chapter first gives the background of the present dissertation. First, the chapter summarizes just-in-time (JIT) compilation, which is a fundamental technology throughout the present dissertation, and it summarizes the two major JIT compilation strategies in JIT compilation while providing an explanation of the advantages and disadvantages of the two JIT compilation strategies (Section 2.1). Then, it explains meta-JIT compilation and a meta-JIT compiler framework (Section 2.2).

2.1 JIT Compilation

A JIT compiler [Aycock, 2003] is a dynamic translator that converts frequently executed code parts, known as the "hot spot," into another language. Typically, the target language is a lower-level one like native code, where the compiler aims to improve the performance. Some general-purpose JIT compilers translate code parts in a source language into native code. Of course, a target language is not limited to native code; the target language can be WebAssembly, LLVM IR, C, or the same language. Figure 2.1 illustrates an overview of how a JIT compiler works on a VM. Basically, a JIT compiler works together with an interpreter and a profiler. First, a source program is executed on an interpreter. While executing the source program, a profiler monitors which code parts are hot. A JIT compiler translates the hot spot into native code. When the control executes the compiled parts again, it returns to native code after the compilation finishes.

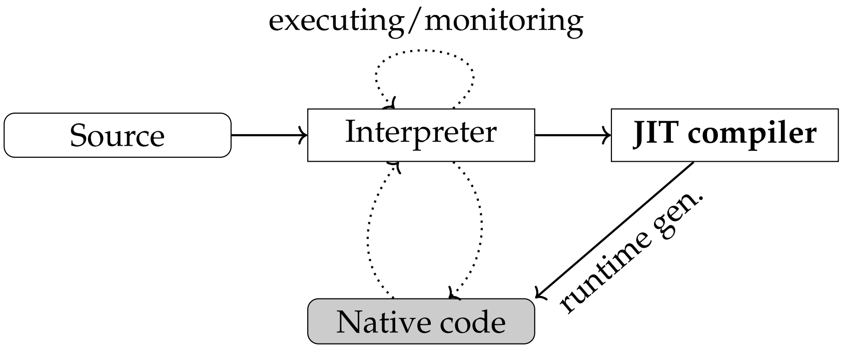


FIGURE 2.1: An overview of how a JIT compiler works in a VM.

With respect to a static (or ahead-of-time, AOT) compiler, it needs to compile all pass paths. This approach can produce fast and executable code, but it includes infrequently-executed code parts. On the contrary, a JIT compiler can selectively translate only hot sections into native code (the other sections are interpreted). This helps reduce the code size and improve space efficiency. Also, a JIT compiler can use runtime information to optimize code, so it is capable of aggressively applying many optimizations to code for further speedup.

A JIT compilation system performs three steps [Aycock, 2003]:

Invocation. A JIT compiler is explicitly invoked when needed. This invocation occurs at the time when a hot spot is found or when a user explicitly requests to use a JIT compiler.

Executability. A JIT compiler typically runs two types of languages: a user language that is translated and a target language that the code is translated into. A JIT compiler should output an executable code; the behavior before and after translation is the same.

Concurrency. A JIT compiler performs code translation concurrently with normal program execution. This concurrency is achieved to invoke JIT compilation via subroutine call, message transmission, or transfer of control to a loop. Also, it can be done in a separate thread or process.

The present dissertation mainly focuses on the invocation and executability, so the concurrency is out of the scope.

2.1.1 JIT Compilation Strategies

A JIT compilation policy can be classified based on the granularity of its compilation unit. This classification is shown in Table 2.1. The strategies range from method-based (Smalltalk-80 [Deutsch and Schiffman, 1984], SELF [Ungar and Smith, 1987], HotSpot [Paleczny, Vick, and Click, 2001]), trace-based (Dynamo [Bala, Duesterwald, and Banerjia, 2000], TraceMonkey [Gal et al., 2009], PyPy [Rigo and Pedroni, 2006]), region-based (HHVM [Ottoni, 2018], IBM's Java JIT [Suganuma, Yasue, and Nakatani, 2006], IMPACT [Hank,

policy	Compilation unit	Implementation
Method-based	Method or function	Smalltalk-80 [Deutsch and Schiffman, 1984], SELF [Ungar and Smith, 1987], HotSpot [Paleczny, Vick, and Click, 2001], GraalVM [Würthinger et al., 2012]
Region-based	Arbitrary region	HHVM [Ottoni, 2018], IBM's Java JIT [Suganuma, Yasue, and Nakatani, 2006], IMPACT [Hank, Hwu, and Rau, 1995]
Basic block-based	Basic block	Higgs [Chevalier-Boisvert and Feeley, 2015, 2016], YJIT [Chevalier-Boisvert et al., 2021], QEMU [Bellard, 2005]
Trace-based	Execution path	Dynamo [Bala, Duesterwald, and Banerjia, 2000], TraceMonkey [Gal et al., 2009], PyPy [Rigo and Pedroni, 2006]

TABLE 2.1: A survery on JIT compilation strategies and their compilation units and implementations.

Hwu, and Rau, 1995]), and basic block (YJIT [Chevalier-Boisvert et al., 2021], QEMU [Bellard, 2005]).

In particular, method-based and trace-based compilation strategies are the most popular. The method-based policy is widely used in VM such as Java virtual machine (JVM), for example, OpenJDK, and OpenJ9, JavaScript virtual machines, for example, V8, SpiderMonkey, and JavaScriptCore. Additionally, the trace-based policy became popular about 10 years ago. Recently, deep learning frameworks, the usefulness of a trace-based policy has been seen: for example, PyTorch provides a trace-based JIT compiler called Pytorch trace JIT [1] to optimize Python code that uses the Pytorch framework.

The difference between the two strategies is the compilation unit. We describe this difference in the following sections.

2.1.2 Method-Based JIT Compilation

Traditional JIT compilers, such as Smalltalk-80 [Deutsch and Schiffman, 1984], SELF [Ungar and Smith, 1987], and Java Virtual Machine (JVM) [Paleczny, Vick, and Click, 2001], employ method-based compilation; they compile a method into native code. A method-based JIT compiler identifies frequently-executed methods from profiled data, compiling that area into native code. Several efficient optimizations, such as inlining, loop unrolling and peeling, and escape analysis, are applied to the native code so that the JIT compiler can produce the code running faster than an interpreter execution.

---

[1] `https://pytorch.org/docs/stable/jit.html`

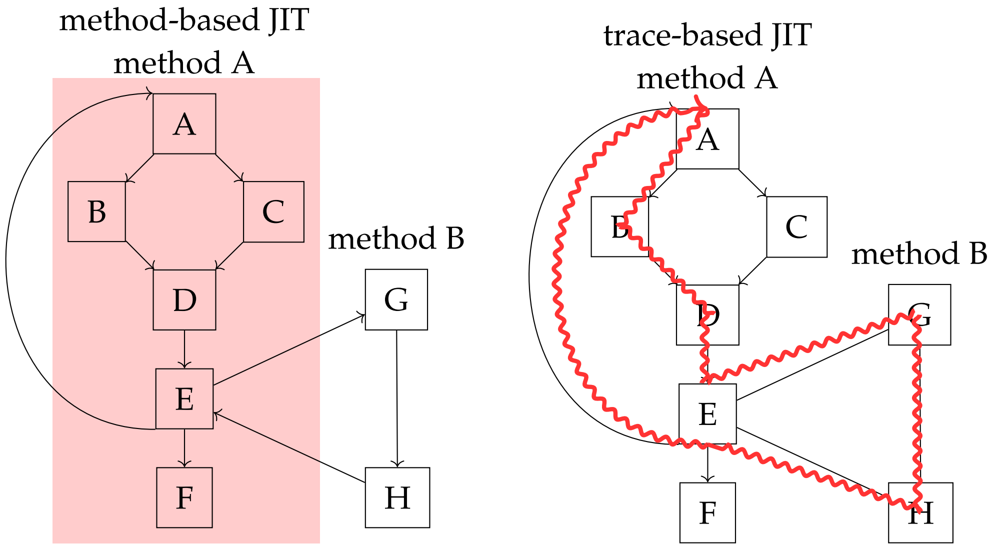


FIGURE 2.2: The difference between method- and trace-based JIT compilation strategies. The linked control flow and blocks represent a method. Areas painted or lines drawn represent a compilation target.

A method-based JIT compilation performs dynamic translation based on a method invocation. The left-hand side of Figure 2.2 shows how a method-based compilation performs. When the number of invoking Method A (A – B – C – D – E – F) exceeds a certain threshold, a method-based JIT is used for compiling. Then, when the number for invoking Method B (G – H) exceeds a certain threshold, it also compiles Method B. Whether inlining is performed depends on the compiler's preference.

2.1.3 Trace-Based JIT Compilation

Trace-based optimization was initially researched by the Dynamo project [Bala, Duesterwald, and Banerjia, 2000], and its technique was adopted to implement compilers for numerous languages such as Lua [Pall, 2005], JavaScript [Gal et al., 2009], Java trace-JIT [Gal, Probst, and Franz, 2006; Inoue et al., 2011], and the SPUR project [Bebenita et al., 2010].

A trace-based JIT compiler basically compiles a loop in a program. "Trace" is a straight line of a program, so a path actually executed is selected and translated into native code. Furthermore, an invocation of a method (or a function call) is automatically inlined. The right-hand side of Figure 2.2 shows how trace-based JIT compilation works. If a hot path is A – B – D – E – (method invokation) – G – H – (return) –E – A, a trace-based JIT compiles that path.

To ensure that the condition in tracing and execution is the same, a *guard* code is placed at every possible point (e.g., if statements) leading in another direction. The guard checks to determine whether the original condition is still valid. If the condition is false, the execution in the native code is stopped

```
def strange_num(a):
  if a % 41 == 0:
    return a * 41
  else:
    return a

def f(n):
  i = 0
  for j in range(n):
    i += strange_num(j)
  return i

f(100000)
```

```
# corresponding trace:
loop_header(i0, j0)
j1 = int_mod(j0, Const(41))
j2 = int_eq(j1, Const(0))
guard_false(j2)
j3 = int_add(j0, Const(1))
i1 = int_add(i0, j3)
jump(i1, j3)
```

Figure 2.3: A simple Python program and its' recorded trace.

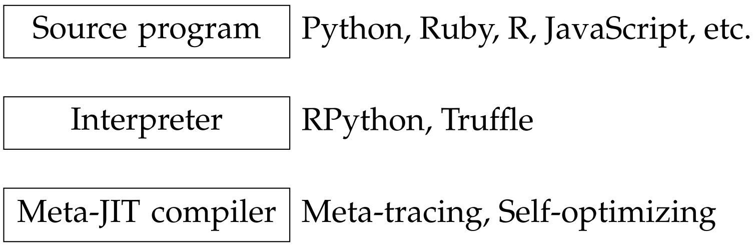

Source program
Python, Ruby, R, JavaScript, etc.
Interpreter
RPython, Truffle
Meta-JIT compiler
Meta-tracing, Self-optimizing

Figure 2.4: System structure of a meta-JIT compiler framework.

and continues to execute, falling back to the interpreter. This behavior is called a guard failure.

As shown in the simple example in Figure 2.3, the hot spot is the `for` loop in the function `f`, so this part becomes a trace. Then, this trace is converted into native code and executed. Note that the function call to `strange_num` is inlined, and the `if` statement is converted into a `guard_false` instruction. The `guard_false` instruction checks whether the variable `j2` is false. If `j2` is false, execution continues. However, if `j2` is true, the execution is stopped, and other programs are interpreted.

2.2 Meta-JIT Compiler Framework

JIT compilation impacts the performance improvement of a VM, but implementation requires time-consuming and error-prone engineering tasks for language developers. Recently, some researchers proposed a *meta-JIT compiler framework* to overcome this problem. A meta-JIT compiler framework provides a convenient and effective way to implement a high-performance VM empowered by a JIT compiler.

The structure of a meta-JIT compiler framework is shown in Figure 2.4. A meta-JIT compiler framework provides a meta-language ((RPython [Bolz et al., 2009] and Truffle/Graal [Würthinger et al., 2012])) to implement an interpreter (Python, Ruby, R, JavaScript, etc.). A source program is implemented with the meta-language. A meta-JIT compiler compiles a source program code by specializing in an interpreter.

The advantage of using a meta-JIT compiler framework is that it makes it easier to build a language without having to construct a compiler for every language from scratch. In general, today's VMs are written in C to achieve their high performance, but a language implementer should be aware of memory management, the threading model, or object layout. Furthermore, to improve the runtime performance, developers usually write a JIT compiler for their VMs by hand. This approach requires the complicated, error-prone, and difficult tasks of language developers. On the contrary, using a meta-JIT compiler can make this task easier. By using a meta-JIT compiler framework, the interpreter can be written in a high-level language such as Python or Java. As proof of this feature, many languages have been implemented with a meta-JIT compiler framework, such as Python, Ruby, R, JavaScript, Smalltalk, PHP, and so on.

There are two approaches in meta-JIT compiler frameworks: partial evaluation-based and meta-tracing-based. The following sections will provide an overview of the two frameworks.

2.2.1 Self-Optimizing Interpreter

A practical method-based (or partial evaluation-based) meta-compilation was recently proposed by [Würthinger et al., 2012, 2017] called as a *self-optimizing interpreter*. This approach is used for implementing numerous source languages such as Ruby [Oracle Lab., 2013], R [Oracle Lab., 2015], JavaScript [Ocacle Lab., 2019], and Smalltalk [Niephaus, Felgentreff, and Hirschfeld, 2019].

Interpreter Definition in a Self-Optimizing Interpreter

Truffle/Graal requires an interpreter based on an abstract syntax tree (AST) to perform its meta-compilation. A defined AST node is responsible for the semantics of a source language (`while`, `if`, etc). There is a common abstract node in Truffle/Graal, and every node inherits this. This execution of a method is carried out by evaluating its root AST node. An AST node has information about local variables and contextual information needed for execution. It has a pointer to the parent node for easy replacement of that node.

A simple definition of the AST node as modeled by Truffle/Graal is shown in Figure 2.5. A node has the `execute` method that takes an argument `f`. The return types of `execute` and `f` are Java's `Object` and `Object[]`, respectively.

The control structure is implemented by using Java's control structure. Basically, non-local returns, that is a transition from a deeply nested node to an

```java
abstract class Node {
  // Executed the operation encoded by this node
  // and returns the result.
  // A frame has information associated with local
  // variables and other contextual information, e.g.,
  // the callee frame.
  public abstract class Object execute(Frame f);

  // Link to the parent node and utility to replace
  // a node in AST.
  private Node parent;
  protected void replace(Node newNode);
}
```

FIGURE 2.5: Definition of an AST node base class, which is retrieved from [Würthinger et al., 2012]

outer node, are implemented with Java's exception. The upper half of Figure 2.6 shows a simplified definition of the `while` block of a source language. The `while` loop in a source language is correlated to Java's `while`, and the non-local return is represented by Java's exception and `try-catch`.

Meta-Compilation in Self-Optimizing Interpreter

The meta-JIT compiler translates abstract-syntax-tree (AST) nodes selected by a partial evaluator into native code. Basically, Truffle/Graal rewrites a generic AST node to a specific node as its meta-compilation. The rewritten node is specialized so that Truffle/Graal can speed up the source program.

Truffle/Graal basically specializes in an AST node by using type information gained from runtime feedback. To perform AST rewriting, Truffle/Graal transforms an AST node that contains a single `execute` method handling all cases into several type-specialized nodes. Each transformed nodes handles a subset of type-specific cases that include the checks for input types. If the input type is different from the expected one, Truffle/Graal replaces the node with a new one that can handle the new case. The AST rewriting transition between different nodes is uniderectional.

Figure 2.7 shows how an AST node is replaced for the plus operator that is used in the `add` function, which is shown in Figure 2.8, here based on the input types. At the first execution, Truffle/Graal rewrites the AST node based on runtime information. The `add` function is written in a dynamically typed language, and the transition starts from an *uninitialized* state. As long as both arguments turn out to be *integer* and there is no overflow in the addition, Truffle/Graal transforms the node into an `integer`-specific one. Otherwise, Truffle/Graal transforms it into the `double`-specific node. If both arguments are `string`, the node is transformed into a string-specific one optimized for the case of string concatenation. If both arguments have different types, the node is transformed to the `generic` one.

```
class WhileNode extends Node {
  protected Node condition;
  protected Node body;

  public Object execute(Frame frame) {
    try {
      while (condition.execute(frame)) {
        try {
          body.execute(frame);
        } catch (ContinueException ex) {
          // Continue in the loop
        }
      }
    } catch (BreakException as) {
      // Breaking out the loop
    }
    return null;
  }
}
```

FIGURE 2.6: Implementation of a `while` block, which is retrieved from [Würthinger et al., 2012].

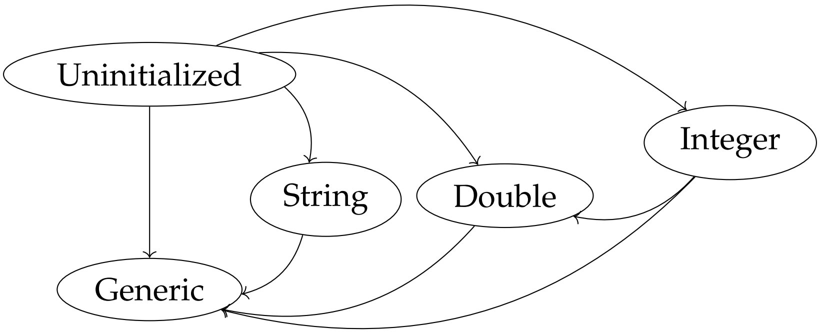


FIGURE 2.7: A transition between different types in an AST tree writing, which is retrieved from [Würthinger et al., 2012].

```
func add(a, b) {
  return a + b;
}
```

FIGURE 2.8: An example of function `add` in a dynamic language.

In addition to an input-type specialization, Truffle/Graal performs a specialization based on return types. This approach is also useful to avoid boxing, which is an approach to wrapping a primitive value (e.g., integer, double, etc.) with a small boxed object. This technique is needed to handle the reference and primitive types at the same time. The common approach for this problem is *tagging*, which is using a bit-level flag in a stored object to check whether primitive or reference variables. However, this approach requires an

```
abstract class Node {
  public abstract Object execute(Frame f);
  public abstract int    executeInt(Frame f)
      throws UnexpectedResultException;
  public abstract double executeDouble(Frame f)
      throws UnexpectedResultException;
  /// ..
}
```

FIGURE 2.9: Type-specialized execute methods, which are cited from [Würthinger et al., 2012].

```
class IntegerAddNode extends BinaryNode {
  Node left;
  Node right;
  public int executeInt(Frame frame)
    throws UnexpectedResultException {
    int a;
    try {
      a = left.executeInt(frame);
    } catch (UnexpectedResultException ex) {
      // Rewrite this node and execute the rewritten node
      // using already evaluated left and right
    }
    int b;
    try {
      b = left.executeInt(frame);
    } catch (UnexpectedResultException ex) {
      // Rewrite this node and execute the rewritten node
      // using already evaluated right
    }
    return a + b;
  }
}
```

FIGURE 2.10: Integer add node using return type specialization as cited in [Würthinger et al., 2012].

additional check for the flag/set a flag every time an object is loaded/stored. Therefore, Truffle/Graal employs a return-type specialization.

The `execute` method shown in Figure 2.9 is generic. According to the return type, different methods such as `executeInt` and `executeDouble` are provided by the abstract `Node` class.

Figure 2.10 illustrates a node that performs integer addition. This node corresponds to the plus operator of the function `add` in Figure 2.8. The method is expected to take two integer arguments and return an integer value. If this assumption turns out to be correct based on the runtime feedback, this method is transformed to a type-specific one. In particular, an uninitialized `AddNode` is rewritten for `IntegerAddNode`. Instead of a generic `execute` method, the specialized method `executeInt` performs the addition. Otherwise, the node

```
// Using Truffle DSL
class WhileNode extends Node {
  // LoopNode class is provided by Truffle DSL
  TruffleDSL.LoopNode loopNode;

  public WhileNode(Node condition, Node body) {
    this.loopNode = TruffleDSL.createLoopNode(condition, body);
  }

  public object execute(Frame f) {
    loopNode.execute(Frame);
  }
}
```

FIGURE 2.11: Implementation of a `while` block using the Truffle DSL

.

```
// An abstract class for binary operators.
// Instead of manually declaring fields, `NodeChild` tells the
// preprocessor to generate a `Node` with two fields `leftNode`
// and `rightNode`.
@NodeChild("leftNode")
@NodeChild("rigthNode")
public abstract class BinaryNode extends Node {}
```

FIGURE 2.12: A binary node that is annotated by `NodeChild`.

```
// An integer-specific add node
class AddNode extends BinaryNode {
  // Integer-specific add method annotated with `Specialization'.
  // Truffle DSL generates several `execute' method from this definition.
  @Specialization(rewriteOn = ArithmeticException.class)
  public int addInt(int left, int right) {
    return left + right;
  }

  @Specialization
  public int add(double left, double right) {
    return left + right;
  }
}
```

FIGURE 2.13: A node representing `add`.

rewrites itself to a generic one and throws an unexpected exception to its callee. This exception is thrown when the associated node is already evaluated, so `executeInt` needs to use the boxed result in the following calculation.

Hint Instructions in Truffle

Truffle DSL The way to implement the AST rewriting, which is explained in Section 2.2.1, requires an additional implementation burden on the language developer. To reduce the effort, Truffle/Graal provides a hint instruction called Truffle DSL, which helps implement a language with Truffle/Graal. The significant advantage is that we no longer manually write many of the boilerplate codes. For example, `WhileNode` can be easily defined as shown in Figure 2.11 by using the Truffle DSL. At the constructor of this node, `loopNode` is created by calling the `createLoopNode` method defined in the Truffle DSL. The `execute` method in `WhileNode` simply invokes the `execute` method defined in `LoopNode`. Inside the Truffle/Graal, `WhileNode` is converted into the one defined in Figure 2.6.

There are two important annotations in the Truffle DSL: `NodeChild` and `Specialization`. `NodeChild` tells a preprocessor to generate a fields with the given name. In Figure 2.12, the abstract `BinaryNode` class is annotated by `@NodeChild("leftNode")` and `@NodeChild("rightNode")`. After preprocessing, the abstract `BinaryNode` has two field declarations named `leftNode` and `rightNode` that are then generated.

The methods should have the `Specialization` annotation[2] placed on them. This annotation allows for customizing an AST generated with input parameters, as well as telling the preprocessor to generate `execute` methods. For example, it is possible to define a customization that allows a fallback when an exception occurs. Figure 2.13 gives an example of the `Specialization` annotation. All methods are annotated by `Specialization`, and `addInt` has only an input value for `ArithmeticException` in the parameter `rewriteOn`. This customization tells Truffle/Graal to fall back to another method like `add` when an arithmetic exception happens when evaluating `addInt`.

Truffle Host Inlining

An AST interpreter written with the Truffle framework is ready for self-optimization. The boundaries of inlining performed by self-optimization can be controlled by the following [Oracle Lab., 2022]:

- blocks calling the method `CompilerDirectives.transferToInterpreter()`
- methods annotated by `TruffleBoundary`

An example of an AST node using the above two features is shown in Figure 2.14. This node shown in Figure 2.14 performs addition between the given two nodes: `left` and `right`. When the self-optimizing system compiles this node, it tries to execute `a = left.executeInt(frame)`. If this execution throws the exception `UnexpectedResultException`, this means that the left node is no longer executed in the fast path. To move back to a slow path, which means an interpreter, `transferToInterpreter()` should be called. The annotation `TruffleBoundary` works like `transferToInterpreter()`. If the method `executeTruffleBoundary` is executed, this method is not inlined

---

[2] https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/dsl/Specialization.html

```
class IntegerAddNode extends BinaryNode {
  Node left;
  Node right;
  public int executeInt(Frame frame)
    throws UnexpectedResultException {
    int a;
    try {
      a = left.executeInt(frame);
    } catch (UnexpectedResultException ex) {
      // executes the following operation in a slow path
      CompilerDirectives.transferToInterpreter();
      // do stuff
    } catch ()
    int b;
    try {
      b = left.executeInt(frame);
    } catch (UnexpectedResultException ex) {
      // executes the following operation in a slow path
      CompilerDirectives.transferToInterpreter();
      // do stuff
    }
    return a + b;
  }

   @TruffleBoundary
   public Object executeTruffleBoundary(Frame frame) {
     // do stuff
   }
}
```

FIGURE 2.14: Example of a bytecode interpreter written in Truffle.

because of the annotation `TruffleBoundary` (the body is executed in an interpreter).

2.2.2 Meta-Tracing JIT Compiler

A trace-based meta-compilation, that is, *meta-tracing*, is the first practical attempt for general interpreters. This technique was established by the PyPy project [Rigo and Pedroni, 2006], which aimed to provide a framework to write a dynamic language in a flexible way. It was first used to implement numerous source languages, such as PyPy [Rigo and Pedroni, 2006], JavaScript [Team, 2015], Ruby [Gaynor et al., 2013], and Smalltalk [Felgentreff et al., 2016].

Figure 2.15 provides an overview of RPython's VM generation and JIT compilation flow. The RPython framework compiles an interpreter written in the RPython language into a VM, which includes a trace-based JIT compiler, garbage collector, and so forth, written in C. At runtime, the program P is first interpreted in a generated VM. When a hot loop is detected, the meta-tracing JIT compiler dynamically compiles it into native code L1. The control

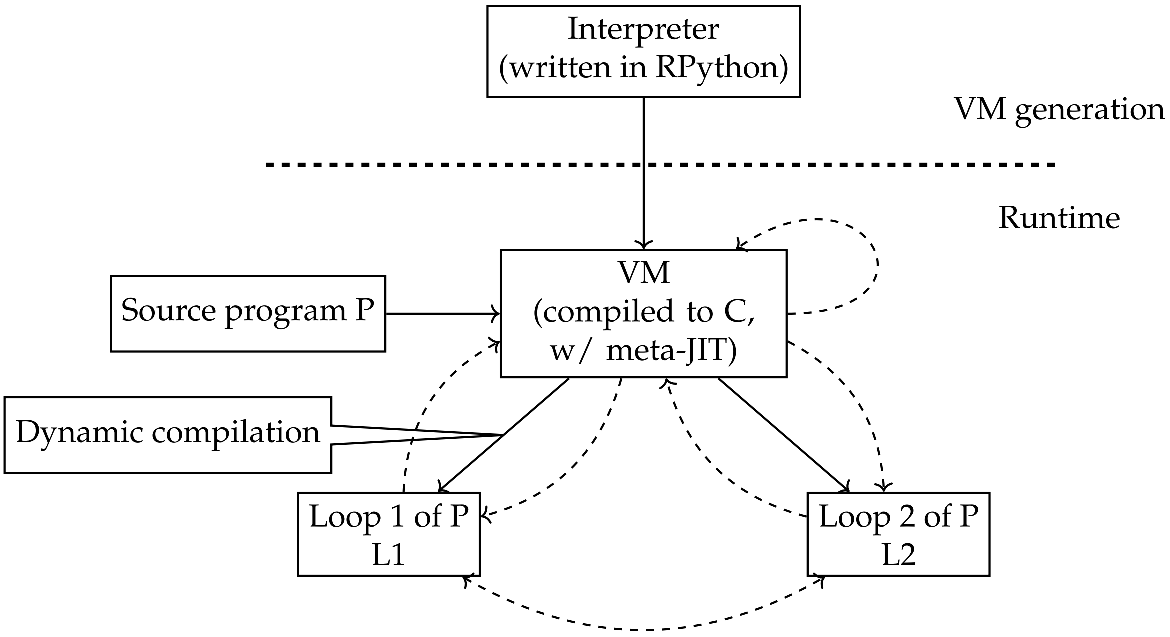


FIGURE 2.15: VM generation and JIT compilation flow in RPython. Solid and dashed lines represent data and control flow, respectively.

goes to L1 after finishing compilation and re-executing the compiled loop. It also jumps to another loop L2 if a destination is already compiled.

Interpreter Definition in a Meta-Tracing JIT Compiler

The RPython framework [Bolz, Diekmann, and Tratt, 2013; Bolz et al., 2009] is a toolchain for creating high-performance VMs powered by a trace-based JIT compiler. For meta-tracing JIT compilation, a language developer needs to write an interpreter definition with the RPython language, which is the restricted subset of Python. The interpreter definition is based on a bytecode format, but Bauman et al. reported that the meta-tracing JIT compilation can effectively be appled to an AST-based interpreter [Bauman et al., 2015].

A RPython-written bytecode interpreter should indicate which part is the loop in a source program. Figure 2.16 shows an example of an interpreter defined by a programming language builder. The example uses two annotations — `jit_merge_point` and `can_enter_jit` — for the identification of a loop. A developer should put `jit_merge_point` at the top of a dispatch loop to identify which part is the main loop, and `can_enter_jit` at the point where a back-edge instruction can occur (where meta-tracing compilation might start).

```
def interp(bytecode):
    stack = []; sp = 0; pc = 0
    while True:
        jit_merge_point(
            reds=['stack','sp'],
            greens=['bytecode','pc'])
        inst = bytecode[pc]
        if inst == ADD:
            v2, sp = pop(stack, sp)
            v1, sp = pop(stack, sp)
            sp = push(stack, sp, v1 + v2)
        elif inst == JUMP_IF:
            pc += 1; addr = bytecode[pc]
            if addr < pc: # backward jump
                can_enter_jit(
                    reds=['stack','sp'],
                    greens=['bytecode','pc'])
            pc = addr
```

FIGURE 2.16: An example interpreter definition written in RPython.

Meta-Compilation in Meta-Tracing JIT Compiler

A meta-tracing JIT compiler traces the execution of an interpreter written in RPython, which is defined by a language builder. The meta-tracing compilation compiles an executed *path* of the source program, even if it has conditional branches. If it does, the compiled code will contain *guards*, each of which is a conditional branch to the interpreter's execution from that point.

Algorithms 1 and 2 illustrate the meta-tracing compilation algorithm in pseudocode. The procedure *JitMetaTracing* takes the following arguments: *rep*—a representation for the interpreter and *states*—the state of the interpreter. A meta-tracing JIT compiler records the execution and checks the operands in the executed operations. It uses the colors *red* and *green* to let the meta-tracing JIT compiler catch runtime information. The color *red* means "a variable in a source language": therefore, the red variables are used to calculate the result of a base program. The color *green* indicates "a variable in an interpreter": then, the compiler will optimize this variable by constant-folding or inlining. If all the operands in one operation are green, the operation is only used for calculation in an interpreter, so the compiler executes it. If at least one variable is red, the compiler recognizes that the operation is in a base program and writes to *residue*.

Hint Instructions in RPython

The RPython framework provides hint instructions to influence the behavior of a meta-tracing JIT compiler.

Algorithm 1: JitMetaTracing(rep, states)

```
input : Representations of an interpreter itself
input : States (e.g., virtual registers and memories) of an interpreter
        itself
output: The resulting trace of the hot spot in a base program
entry_states ← states;
repeat
    residue ← [ ]; // A data to store the result
    op ← rep.current_operation(states);
    if op = conditional branch then
        if op has red variables then
            guard ← op.mk_guard(states);
            residue.append(guard);
        eval(op, states, residue);
    else if op = function call to f then
        inline f;
    else
        eval(op, states, residue);
until op = jit_merge_point ∧ entry_states = states;
return residue;
```

Algorithm 2: Eval(op, states, residue)

```
if op has red variable then
    op.const_fold(states);
    residue.append(op);
else
    op.execute(states);
```

Promotion Promotion is a technique to turn an arbitrary variable in a trace into a constant. When a variable is promoted, a guard instruction is inserted to validate that the value of the variable is constant. Once promotion is successful, the meta-tracing JIT compiler can apply more aggressive constant-folding. However, promoting variables too much may occur in the case of guard failure, generating a large number of bridges from very similar `guard_value` opcodes. The followings are the promotion functions provided by RPython:

- `promote`: promote a variable to a constant
- `promote_string`: same to `promote`, but promote string by value
- `promote_unicode`: same to `promote_string`, but promote unicode string by value

Trace elidable Decorating a function with trace-elidable means that the function returns a same result with identical arguments, having no side-effects. The call to a trace-elidable function is replaced with the result of a trace-elidable call if RPython can completely guess that the result should not change with the same arguments, same numbers, and same pointers. The decorator `@elidable` is used to tell RPython that the decorated function is trace-elidable. The decorator `@elidable_promote` is a special trace-elidable hint function that promotes all arguments to being constant.

Inlining or not inlining function Unrolling a function call or leaving a function is decided by RPython. RPython tries to inline a function as much as possible, but gives up inlining when it would produce too much code. Because unrolling loops can be harmful due to the explosion of trace sizes, the decision is carefully managed in RPython internally. However, it is obvious when a function should be unrolled or not in some case, so RPython provides decorators to tell RPython whether or not to unroll the decorated function. The RPython decorator `@unroll_safe` specifies that the decorated function should always be unrolled. On the other hand, `@dont_look_inside` specifies that RPython should produce a call to the function not trace the inside of the function. Note that `@dont_look_inside` executes the body of the decorated function to trace the successors of the function.

Accessing runtime information RPython provides several functions to allow us that access runtime or tracing-time information. These functions are useful when the user wants to describe behavior in the interpreter that will be executed in specific situations, such as during tracing or after compilation. The function `we_are_jitted` returns True after tracing and translation to C, and False during interpreter execution. Similarly, `we_are_translated` returns True after translation and False otherwise.

Chapter 3

Motivation and Proposal

Contents

3.1 Scope of Compiling a Source Program 25

3.2 Level of Compiling a Source Program 27

3.3 Dilemma: Hard to Extend Generated VMs from a Meta-JIT Compiler Framework 29

3.4 Proposal: Multi-Role Meta-Tracing JIT Compilation 30

- 3.4.1 System Overview: Multi-Role Meta-Tracing JIT Compiler Framework 32
- 3.4.2 Underlying Techniques for Multi-Scope Compilation 33
- 3.4.3 Underlying Techniques for Multi-Level Compilation 35

The goal of the present dissertation is to realize *multi-role meta-tracing compilation*. The multi-role meta-tracing JIT compiler can support multiple scopes of compilation targets and multiple levels of compilation. In addition, the multi-role compilation is performed on top of a meta-tracing JIT compiler framework, so it is possible to generate a multi-role JIT compiler by writing an interpreter.

This chapter illustrates the motivation behind creating multi-role JIT compilation. First, this chapter discusses the major JIT compilation policies in terms of the scope of the compilation in Section 3.1 and the level of the compilation in Section 3.2. Then, Section 3.3 describes the problem of realizing multi-role meta-tracing JIT compilation. Finally, the chapter provides a high-level overview of the multi-role meta-tracing JIT compiler in Section 3.4.

3.1 Scope of Compiling a Source Program

There is a wide range of compilation policies in JIT compilers, and they can be classified around the compilation scope. To begin, a method-based compilation, which configures its compilation scope as a method, is the most traditional and well-studied policy. This policy has been adopted in Smalltalk [Deutsch and Schiffman, 1984], Java [Paleczny, Vick, and Click, 2001], and other VMs. Furthermore, a trace-based compilation, which sets its scope as the execution path in a source program, has been studied since the later half of the 2000s. The concept of trace-based compilation is proposed by Dynamo [Bala, Duesterwald, and Banerjia, 2000], and it has been used in some real-world VMs such as PyPy [Rigo and Pedroni, 2006], TraceMonkey [Gal et al., 2009], and LuaJIT [Pall, 2005]. As mentioned in Sections 2.2.1 and 2.2.2, the two scopes are used in the JIT compilation system of the current meta-JIT compiler frameworks. However, these policies have their own advantages and disadvantages, so it is difficult to pinpoint one or the other as the clear winner. Given this context, compilation policies using the region of a program as a compilation scope, called region-based compilation, is attracting attention recently. This policy is commercially used in HHVM [Ottoni, 2018], which is a VM for PHP and Hack languages built by Meta.

The following paragraphs describe the features, advantages, and disadvantages of each of the compiled scopes in a heavyweight JIT compilation.

Method-based Compilation Policy Method-based compilation [Deutsch and Schiffman, 1984; Paleczny, Vick, and Click, 2001; "The IBM J9 Java Virtual Machine for Java 6" 2009; Ungar and Smith, 1987], or simply speaking method JIT, is a compilation policy that specializes the scope of a frequently executed method and translates it into machine code at runtime. This policy has been a well-researched traditional approach since the 1980s. The advantage of method-based compilation is that it can share many optimization techniques that are used in traditional static compilers. Therefore, it can perform efficient optimizations, including inlining, control flow analysis, loop unrolling and peeling [Paleczny, Vick, and Click, 2001], escape analysis [Choi et al., 1999b], and data flow analysis, and it exhibits good performance on average.

However, careful management of the compilation target is necessary because the compilation unit could cover code parts that are rarely executed. It has the risk of leading to a code size blow-up if attention is not paid to it. When aggressive inlining is applied to a method with deeply nested function calls, the compiled code size increases, leading to a longer compilation time. Therefore, this requires well-planned inlining to a method for reducing the overhead of a method call.[1]

Trace-Based Compilation Policy Trace-based compilation targets the "trace" of the executed code. It can target a larger scope of compilation than method-based JIT compilation, because it is capable of automatically inlining a function call (or method invocation) [Bolz et al., 2011b; Gal et al., 2009]. Because of the compilation of linear execution paths, the trace-based JIT compiler can aggressively apply optimization techniques, including constant subexpression elimination, dead code elimination, constant folding, and register allocation removal [Bolz et al., 2011a]. Thus, this policy obtains better results with specific programs with branching possibilities or loops [Bauman et al., 2015; Huang, Masuhara, and Aotani, 2016]. Moreover, it can execute aggressive function inlining at a low cost, because a trace-based JIT compiler tracks and records the execution of a program so that a resulting trace will include an inlined function call [Gal, Probst, and Franz, 2006]. This reduces the overhead of a function call, creating opportunities for further optimization.

---

[1] To avoid this problem, the developers of JVMs tune their threshold to start JIT compilation by using well-designed benchmark suites such as SPECjvm[2] and DaCapo [Blackburn et al., 2006].

However, when a trace-based JIT compiler traces a complex control flow, the obtained trace will not be useful since the control cannot stay on the trace for long periods of time, and many deoptimizations occur, which means escaping from an interpreter to a trace in the context of tracing JIT compilation, and it leads to bad runtime performance. This problem is called *path-divergence problem* [Hayashizaki et al., 2011; Huang, Masuhara, and Aotani, 2016]. Inoue et al. [Inoue et al., 2011] reported that their trace-based JIT compilation outperformed the method-based JIT compilation in three benchmarks, while it became slower in seven benchmarks in DaCapo. The ratio ranges from 20 % slower to 25 % faster than the method-based JIT compilation.

Region-Based Compilation Policy Region-based JIT compilation has been recently adopted in the second generation of HHVM, which is a production level VM for the PHP and Hack languages. [Ottoni, 2018] argues that a region-based compilation policy is good for dynamic languages. Because its compilation unit is not restricted to entire method bodies, single basic blocks, or straight-line traces, it can provide the flexibility that is important for compilation time and code size. Furthermore, [Suganuma, Yasue, and Nakatani, 2006] stated that region-based compilation can avoid frequent side exists[3] that affect basic block- and trace-based policies.

As described above, each policy has its own advantages and disadvantages. In particular, a multi-scope approach such as region-based compilation is promising because it can take advantage of the two popular compilation scopes–method-based and trace-based. However, the meta-JIT compiler frameworks do not support multi-scope compilation. The reason for this is described in Section 3.3.

3.2 Level of Compiling a Source Program

Real-world applications are divided into sections that are executed frequently, slightly, and not at all. In the present dissertation, frequently executed and slightly executed sections are called hot spots and warm spots, respectively. Real-world managed VMs, such as Hotspot, V8, JavaScriptCore, have multiple levels of compilation consisting of lightweight and heavyweight compilers. The motivation to support multiple levels of compilation is to use lightweight compilers for programs with short execution times and to apply heavy-weight compilers to programs with long execution times. Heavyweight JIT compilers such as the C2 [Paleczny, Vick, and Click, 2001]

---

[3] It means a costly fallback to interpreter execution from machine code execution.

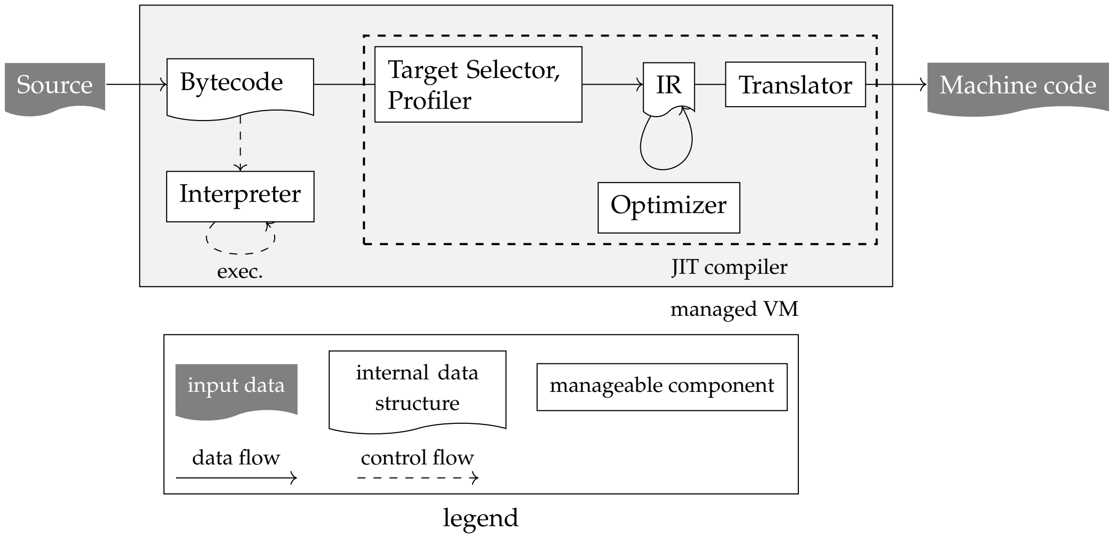

Source
Bytecode
Interpreter
exec.
Target Selector,
Profiler
IR
Translator
Optimizer
JIT compiler
Machine code
managed VM
input data
internal data structure
manageable component
data flow
control flow
legend

FIGURE 3.1: Simplified overview of a managed VM.

in Hotspot, the Turbofan compiler [Google, 2015a] in V8 [Google, 2015b], and the FTL compiler [Pizlo, 2014] in JavaScriptCore [Pizlo, 2020] produce fast machine code and consume compilation time because they use expensive optimization techniques. They are usually applied to hot spots because performance improvement can be expected, even if the cost of the compile time is paid. However, if a heavyweight JIT compiler is used for warm spots, the cost of the compile time is inevitable, because warm spots are executed less often than hot spots. Therefore, warm spots are compiled by lightweight JIT compilers. A lightweight JIT compiler, such as the C1 [Kotzmann et al., 2008] in Hotspot, the Ignition compiler [Google, 2016] in V8, the DFG compiler [Barati, 2022] in JavaScriptCore, quickly generates machine code with inexpensive optimizations. In the DaCapo benchmark that simulates a real-world workload of Java applications, [Blackburn et al., 2006] stated that about 30% of declared methods are compiled by lightweight compilation and 10% of methods compiled by heavyweight compilation.

In contrast to the managed VMs described above, there have been few studies on multilevel compilation on a meta-JIT compiler framework. The two meta-JIT compiler frameworks—RPython and Truffle/Graal—have a limited option to customize their compilation levels. As mentioned in Sections 2.2.1 and 2.2.2, they provide a *hint instruction*, which allows language developers to customize the behavior of their compiler and the code compiled. However, the latter is currently used for ad hoc optimization. The reason and approach for the present dissertation are discussed in Section 3.3.

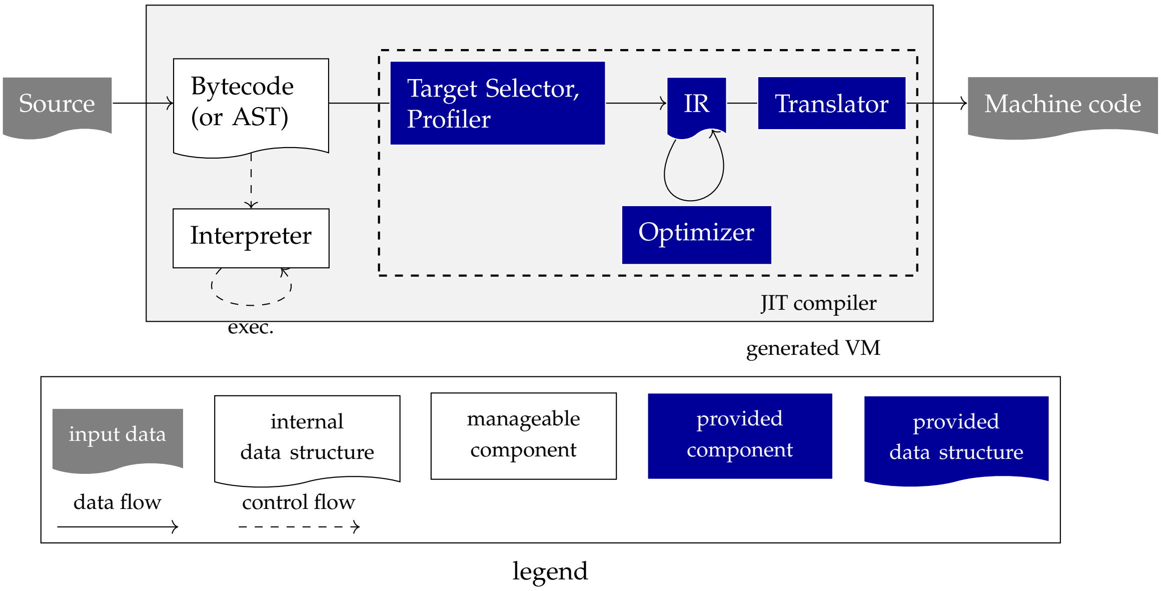

Source
Bytecode (or AST)
Interpreter
exec.
Target Selector, Profiler
IR
Translator
Optimizer
JIT compiler
Machine code
generated VM
input data
internal data structure
manageable component
provided component
provided data structure
data flow
control flow
legend

FIGURE 3.2: Simplified overview of a generated VM from a meta-JIT compiler framework.

3.3 Dilemma: Hard to Extend Generated VMs from a Meta-JIT Compiler Framework

Normally, the components of a VM are managed by the developer (especially, it is called a managed VM here). In other words, the addition of a new functionality can be achieved by making changes to the components of the managed VM or by adding new components. Figure 3.1 shows the simplified overview of the components of a managed VM. For example, when developers create a new JIT compiler with a different compilation scope than the existing one, they modify a target selector/profiler, design a new intermediate representation (IR), and create a new optimizer and translator. In addition, if they add a feature about multilevel compilation, they add a new lightweight compiler in the JIT compiler of a VM. This addition and modification are possible because all components of a VM are manageable.

In contrast to the case of a managed VM, it is difficult to easily add a new compilation, such as multi-scope compilation, to the meta-JIT compiler framework. The dilemma is that it is difficult to add a new feature to the VM generated by the meta-JIT compiler framework that would be possible in a managed VM. This is because, as shown in Figure 3.2, almost all the components of a generated VM are provided by the framework and cannot be freely modified. If developers using a meta-JIT framework tried to add new features to their VMs, they would have to rewrite the framework itself. This approach is not realistic because the implementation costs are too high.

The only component that the developer can freely implement and extend is the interpreter. The meta-JIT compiler frameworks provide hint instructions, such as the `elidable` decorator in RPython, to pass information about the variables or functions to the meta-JIT compiler for further optimization, but they are not currently considered as being useful in implementing a new compilation behavior. If they are organized in a unified way to generate a new compilation behavior along with the existing one, it not only eases the user's implementation burden, but it also allows for the pursuit of performance in the implementing language.

3.4 Proposal: Multi-Role Meta-Tracing JIT Compilation

As in the discussions in Sections 3.1, the two major JIT compilation policies—method- and trace-based compilation policies—have their own advantages and disadvantages. In summary, method-based JIT compilation can perform well on average, but it needs careful management of the compilation code size to avoid code bloat. On the other hand, trace-based JIT compilation can emit highly optimized code by keeping the code size small compared with method-based JIT compilation. However, the performance becomes worse when tracing a program with complex control flow, which is called the path-divergence problem. A region-based compilation policy can have the benefits of both policies and is the focus here.

In addition, as explained in Section 3.2, today's managed VMs support multilevel compilation to achieve optimal performance in all situations. In other words, they have lightweight and heavyweight compilers and use lightweight compilation for short-running scripts and heavyweight compilation for long-running applications.

The goal of the present dissertation is to realize multi-role meta-tracing JIT compilation. However, according to the description in Section 3.3, it is difficult adding new features to generated VMs from a meta-JIT compiler framework because generated VMs have no choice but to use VM components provided by the framework. Therefore, if the development method of the managed VM is changed, it is necessary to create a new meta-JIT compiler framework or make major changes to the existing framework to achieve multi-role compilation. Not only may the implementation cost increase, but if the existing framework is extended, the introduction of new intermediate representations will cause the existing optimizers and transformers to be rewritten.

In order to achieve multi-role meta-tracing JIT compilation, the present dissertation proposes a *hint instruction-based approach*. In the context of multi-role meta-tracing JIT compilation, the hint instruction is a pseudo-function that is inserted into an interpreter. The hint instruction affects the behavior of the meta-tracing JIT compiler by inserting it into an interpreter to be traced. Depending on the definition of hint instructions in an interpreter, the meta-tracing JIT compiler can generate code at multi-compilation-scope and multi-compilation-level.

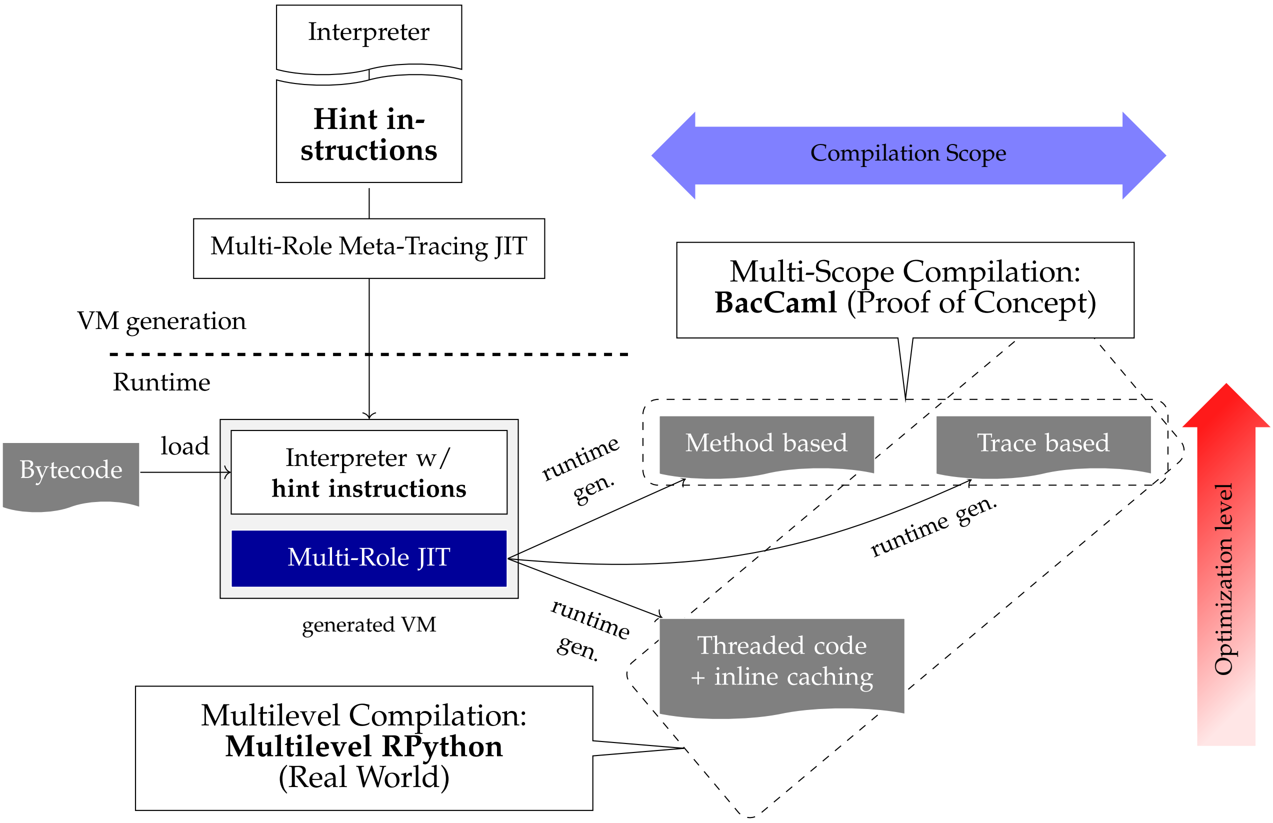

Interpreter
Hint instructions
Multi-Role Meta-Tracing JIT
VM generation
Runtime
Bytecode
load
Interpreter w/ hint instructions
Multi-Role JIT
generated VM
Compilation Scope
Multi-Scope Compilation: BacCaml (Proof of Concept)
runtime gen.
Method based
Trace based
runtime gen.
runtime gen.
Threaded code + inline caching
Multilevel Compilation: Multilevel RPython (Real World)
Optimization level

FIGURE 3.3: Overview of a multi-role meta-tracing JIT compiler framework.

The advantage of this approach in working toward the realization of multi-role meta-tracing JIT compilation is that it can be achieved without drastically modifying a meta-JIT compiler framework itself. The multi-role compiler is achieved by using the characteristics of meta-tracing JIT compilation. In other words, a meta-tracing JIT compiler executes a bytecode interpreter that is written by the user during the tracing, and with this, it is possible to change the behavior of the tracer and the form of the compiled code it generates, here depending on the hint instructions that are inserted into the interpreter.

The multi-role JIT compilation has two aspects: multi-scope compilation and multi-level compilation. Multi-scope compilation can take multiple compilation scopes when it performs JIT compilation. In summary, it can apply method-based compilation to a function `f` and trace-based compilation to a function g. The method-based compilation is realized by (meta-)trace-based compilation with hint instructions. As a proof of concept, this dissertation proposes the hint instructions and interpreter design for multi-scope compilation and *BacCaml* framework [Izawa and Masuhara, 2020] in Chapter 4. Furthermore, the other aspect is that multilevel compilation has multiple compilation levels: lightweight and heavyweight. The multilevel compilation is implemented in RPython, a real-world meta-tracing

JIT compiler framework. It supports method-based threaded code generation [Izawa et al., 2022] as lightweight compilation and trace-based compilation as heavyweight compilation. Chapter 5 describes the details of *Multilevel RPython*, which supports such multi-level compilation. In particular, the chapter introduces a way to implement an optimization technique called inline caching [Deutsch and Schiffman, 1984] by using hint instructions for lightweight compilation in Section 5.4.

3.4.1 System Overview: Multi-Role Meta-Tracing JIT Compiler Framework

This section provides an overview of the hint instruction-based approach to multi-role meta-tracing JIT compilation based on Figure 3.3. First, the user generates a VM with multi-role JIT compiler from a written interpreter instrumented with the hint instructions. At runtime, different compilations with different compilation scopes are applied to different parts of a program when the hint instructions tell the multi-role JIT compiler to perform a multi-scope compilation (the details are described in Chapter 4). In the same way, when the hint instructions tell the multi-role JIT compiler to perform a multi-level compilation, it will use a lightweight and a heavyweight compilation (details are shown in Chapter 5).

To switch compilation scopes and levels, the present dissertation implements two strategies: one based on annotation and one on profiling. BacCaml, which is described in Chapter 4, uses the annotation-based strategy, and Multilevel RPython, which is described in Chapter 5, uses the profiling-based strategy. Figure 3.4 illustrates both strategies using pseudocode. The multi-scope compiler generated by BacCaml decides the compilation scope depending on the annotation specified in a function. Figure 3.4a shows the example. The multi-scope compiler performs method-based compilation when the hint instruction `can_enter_method_jit` defined in the interpreter is called. Similarly, it performs trace-based compilation when the `can_enter_tracing_jit` hint is called. Multilevel RPython, on the other hand, uses a profiling-based strategy. Figure 3.4b shows an example using pseudocode. There are two types of JIT drivers: `threaded_driver` for lightweight compilation and `tracing_driver` for heavyweight compilation. The `threaded_driver.can_enter_jit(..)` hint is placed at the start of an interpreter definition, and the `tracing_driver.can_enter_jit(..)` hint is placed at the point where a back-edge jump occurs. Each hint instruction tracks the number of times it is called, and JIT compilation occurs when a threshold has been exceeded. The threshold can be set by the user of the multi-role meta-tracing JIT compiler framework.

```
(* Compiling this function
 * with method-based JIT *)
let%mj rec fib n =
  if n <= 1 then n
  else fib (n-1) + fib (n-2)

(* Compiling this function
 * with trace-based JIT *)
let%tj rec sum n =
  if n <= 1 then n
  else n + sum (n-1)
```

(A.1) Source program.

```
(* compiled bytecode *)
fib:
  METHOD_COMP
  DUP
  CONST_INT 0
  LT
  ...

sum:
  TRACING_COMP
  DUP
  CONST_INT 0
  LT
  ...
```

(A.2) Compiled bytecode.

```
(* interpreter *)
let rec interp stack sp bytecode pc =
  let instr = bytecode.(pc) in
  if instr = METHOD_COMP then
    can_enter_method_jit (..)
    interp stack sp bytecode (pc+1)
  else if instr = TRACING_COMP then
    can_enter_tracing_jit (..)
    interp stack sp bytecode (pc+1)
  else if ...
```

(A) Annotation-based approach to specify a JIT compilation policy.

```
threaded_driver = JitDriver(threaded_code_gen=True, ..)
tracing_driver  = JitDriver(..)

def interp(bytecode):
  pc = 0, stack = []
  threaded_driver.can_enter_jit(..)
  while True:
    instr = bytecode[pc++]
    if instr == JUMP_BACKWARD:
      target = bytecode[pc++]
      pc = target
      tracing_driver.can_enter_jit(..)
    elif ...
```

(B) Profiling-based approach to specify a JIT compilation policy.

FIGURE 3.4: Two approaches to a specify JIT compilation policy in BacCaml and Multilevel RPython.

3.4.2 Underlying Techniques for Multi-Scope Compilation

To achieve multi-scope compilation in meta-tracing JIT compilation, it should establish a way to perform method-based compilation though (meta-)trace-based compilation. This section proposes the hint instructions that enable multi-scope compilation on top of a meta-tracing JIT compiler framework. This approach does not build a new compiler from scratch, but instead, it guides the tracer of a meta-tracing JIT compiler through the use of the hint instructions.

Hint Instructions and Compilation Overview

This section briefly shows the hint instructions and compilation overview of method-based compilation when using a meta-tracing JIT compiler, namely *Method JIT by Tracing*. BacCaml and Multilevel RPython differ in the way Method JIT by Tracing is implemented: BacCaml is implemented by modifying the compiler, whereas Multilevel RPython is implemented by using a new data structure called `traverse_stack`.

The Method JIT by Tracing traces all possible paths of a target method at once with the help of hint instructions.

Hint Instructions The hint instructions that enable a meta-tracing JIT compiler to perform method-based compilation are as follows:

- `traverse_if(cond, other_dest)`: indicates the point where the trace should traverse both sides.
- `traverse_JUMP(dest)` and `traverse_RET(value)`: tell the meta-tracing JIT compiler to backtrack.
- `dont_look_inside`: tells the tracer not to inline this method call.

In addition, these hint instructions are inserted in an interpreter like Listing 1. By using this interpreter definition, a multi-role meta-tracing JIT compiler performs Method JIT by Tracing. The compilation overview is explained below.

Compilation Overview The overview of Method JIT by Tracing's compilation flow is shown in Figure 3.5. Taking the bytecode-formatted method `f` and the interpreter shown in Listing 1 as an example:

1. First, the tracer traces `INT 10000` and `LT`.
2. When tracing `JUMP_IF L1`, the tracer executes `traverse_if`. At this point, the tracer saves its state (this state will be rewound later) and trace one side.
3. Next, the tracer traces `INC`.
4. When tracing `JUMP_BACK L0`, the tracer executes `traverse_JUMP(target)`.
    (a) The tracer leaves the jump instruction to `target`, which corresponds to `L0` in bytecode, hence resulting in the trace.
    (b) Then, it backtracks to `JUMP_IF L1` and rewinds the saved states.

```
while True:
  instr = bytecode[pc++]
  if instr == JUMP_IF:
    target = bytecode[pc++]
    traverse_if(stack.pop(), pc+2):
      pc = target
  elif inst == JUMP:
    target = bytecode[pc++]
    traverse_JUMP(target)
  elif instr == CALL:
    r = dont_look_inside(interp(..))
    stack.push(r)
  elif instr == RET:
    v = stack.pop()
    traverse_RET(v)
  elif instr == INT:
    n = bytecode[pc++]
    stack.push(n)
  elif instr == INC:
    x = stack.pop()
    stack.push(x + 1)
  elif instr == LT:
    y, x = stack.pop(), stack.pop()
    if x < y:
      stack.push(True)
    else:
      stack.push(False)
```

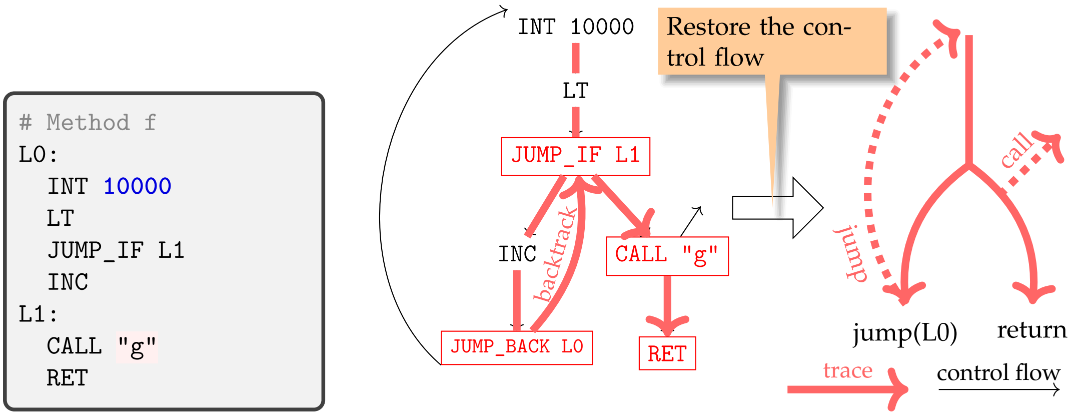
LISTING 1: Brief example of an interpreter instrumented with the hint instructions for Method JIT by Tracing.

5. When tracing `CALL "g"`, the tracer executes `dont_look_inside`. Then, it does not track the call destination but leaves a call instruction to the function g.
6. Finally, when tracing `RET`, the tracer executes `traverse_RET(v)`. Then, it emits `return v` to the resulting trace and finishes the trace.

The temporal output trace does not retain the original control flow because the tracer jumps back to `JUMP_IF L1` at `JUMP_BACK`. Thus, the compiler cuts/stitches the temporal output and restores the original control flow. In BacCaml, this is done during tracing, but in Multilevel RPython, it is done after traversing a method (this is called *trace-stitching*).

3.4.3 Underlying Techniques for Multi-Level Compilation

As described in Section 3.4.2, it is possible to create a new compilation behavior by instrumenting an interpreter with hint instructions. The key idea behind implementing multi-level compilation is to prepare different interpreters with different hint instructions. For example, if you want to implement a JIT compiler with two levels of compilation, you need to instrument one interpreter with hint instructions for lightweight compilation and one



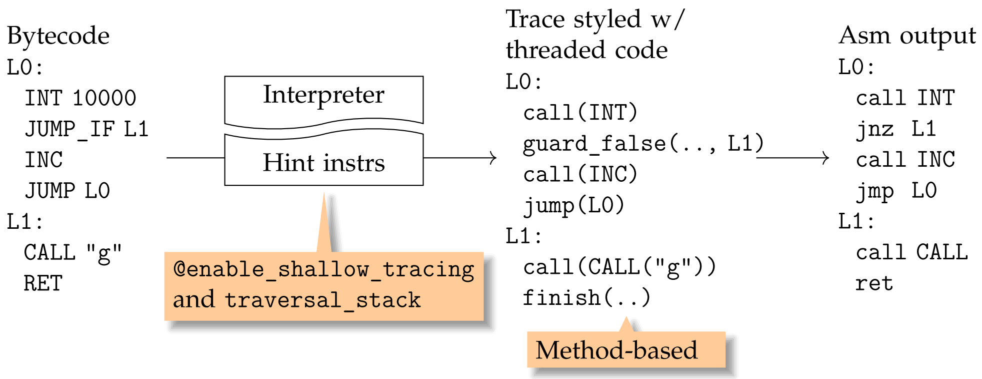
```
# Method f
L0:
  INT 10000
  LT
  JUMP_IF L1
  INC
L1:
  CALL "g"
  RET
```

INT 10000
LT
JUMP_IF L1
INC
CALL "g"
JUMP_BACK L0
RET
backtrack
Restore the control flow
jump
call
jump(L0)
return
trace
control flow

(A) Source program as bytecode format.

(B) Overview of how Method JIT by Tracing works on the example. The tracer first traverses all possible paths of the method f. Then, the compiler restores the original control flow.

FIGURE 3.5: Compilation Overview of Method JIT by Tracing.



```
Bytecode
L0:
  INT 10000
  JUMP_IF L1
  INC
  JUMP L0
L1:
  CALL "g"
  RET
```

```
Trace styled w/
threaded code
L0:
  call(INT)
  guard_false(.., L1)
  call(INC)
  jump(L0)
L1:
  call(CALL("g"))
  finish(..)
```

```
Asm output
L0:
  call INT
  jnz  L1
  call INC
  jmp  L0
L1:
  call CALL
  ret
```

Interpreter
Hint instrs
@enable_shallow_tracing
and traversal_stack
Method-based

FIGURE 3.6: Compilation overview of lightweight compilation in Multilevel RPython.

interpreter with hint instructions for heavyweight compilation. The hinting for heavyweight compilation is already provided by RPython, the question is how to achieve lightweight compilation with hinting.

To achieve multilevel compilation in a multi-role meta-tracing JIT compiler framework, the present dissertation proposes the hint instruction called `@enable_shallow_tracing` for lightweight compilation. With the help of `@enable_shallow_tracing`, the multi-role meta-tracing JIT compiler can emit threaded code [Bell, 1973]. Figure 3.6 shows the compilation overview of lightweight compilation in Multilevel RPython, and Listing 2 illustrates the interpreter definition instrumented using `@enable_shallow_tracing`. Note that `@enable_shallow_tracing` decorates every handler so that it emits threaded code. In addition, with the help of `traverse_stack`, which is described in Section 5.2.3, the scope of the output trace is method-based. The technical details are described in Chapter 5.

```
# Interpreter for lightweight compilation
while True:
  instr = bytecode[pc++]
  if instr == INC:
    handler_INC(stack)
  elif instr == CALL:
    handler_CALL(stack)
  elif ...
    ...

# instrumented with the hint instructions
@enable_shallow_tracing
def handler_INC(stack):
  x = stack[sp--]
  z = add(x, 1)
  stack[sp++] = z

@enable_shallow_tracing
def handler_CALL(stack):
  r = interp(stack, ..)
  push(r)
```

LISTING 2: Brief example of interpreter definition for lightweight compilation.

Chapter 4

BacCaml – a Proof-of-Concept Multi-Scope Meta-Tracing JIT Compiler

Contents

4.1 Introduction 39

4.2 Mixing The Two Compilation Strategies in Meta-Level . . 41

4.2.1 Method-Based Compilation by Tracing 41

4.3 Stack Hybridization 46

4.3.1 Combination Problem 46

4.3.2 Bridging Native Code with Different Calling Conventions 48

4.4 Evaluation 49

4.4.1 Setup 50

4.4.2 Standalone JIT Microbenchmark 51

4.4.3 Multi-Scope JIT Experiment 53

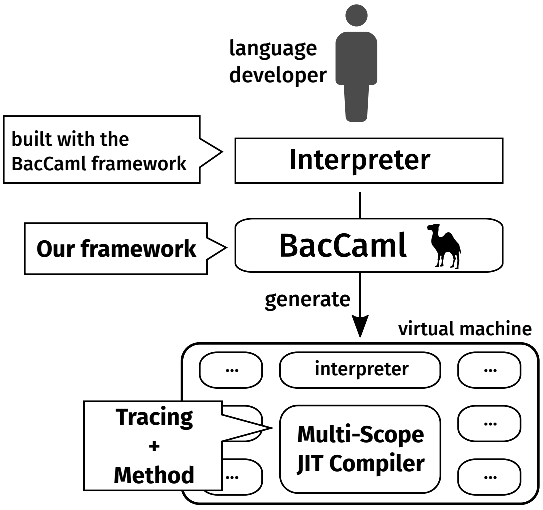
4.5 Related Work 55

4.6 Conclusion 57

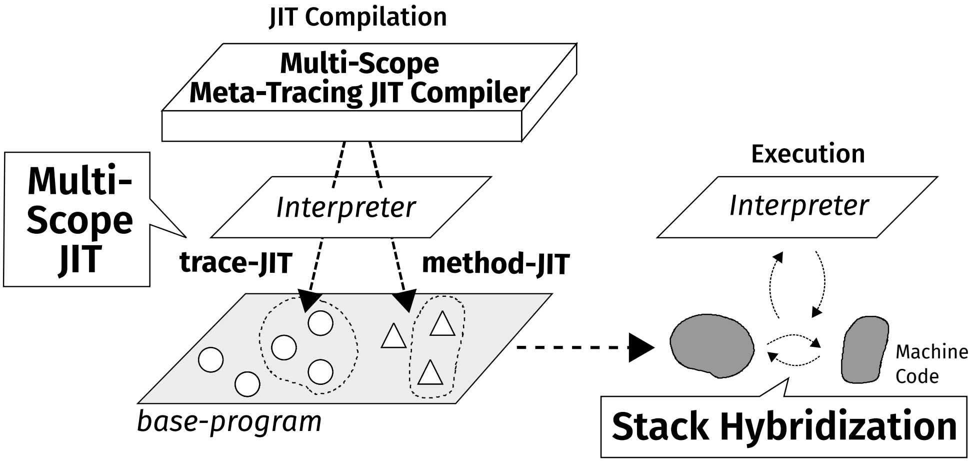
4.1 Introduction

This chapter evaluates the performance of BacCaml's trace-based and method-based compilations. It first briefly introduces the current status of BacCaml, and how we took the data of microbenchmark programs. Next, it shows the results of evaluation for BacCaml by running microbenchmark programs.

The basic idea of achieving multi-scope compilation is realizing method-based compilation by extending a (meta-) tracing compiler and mixing them. Since a trace has no control flow and inlines a function, we create our method-based compilation by customizing the tracing JIT compilers' features


language developer
built with the BacCaml framework
Interpreter
Our framework
BacCaml
generate
virtual machine
...
interpreter
...
Tracing + Method
Multi-Scope JIT Compiler
...
...
...
...

(A) A virtual machine generation with a multi-scope meta-tracing JIT compiler framework.


JIT Compilation
Multi-Scope Meta-Tracing JIT Compiler
Multi-Scope JIT
Interpreter
trace-JIT
method-JIT
base-program
Execution
Interpreter
Machine Code
Stack Hybridization

(B) A runtime overview of a generated multi-scope JIT compiler.

FIGURE 4.1: The overviews of a multi-scope meta-tracing JIT compiler framework.

to cover all the paths in a method. Therefore, the multi-scope meta-tracing JIT compiler framework shares almost all its implementations between the two trace- and method-based compilers. Details are explained in Section 4.2.1.

In addition to the leverage strength of the two JIT compilation strategies, a multi-scope compilation aims to resolve the path-divergence problem by selectively applying method-based compilation to the functions that cause the problem, and applying trace compilation to the other parts of a program. Since this proposal focus es on combining the different two strategies, dynamically selecting a suitable strategy depending on target programs' structure is left as future work.

The basic idea of achieving multi-scope compilation is realizing method-based compilation by extending a (meta-) tracing compiler and mixing them. Since a trace has no control flow and inlines a function, we create our method-based compilation by customizing the tracing JIT compilers' features to cover all the paths in a method. Therefore, our multi-scope meta-tracing JIT compiler framework shares its implementations between the two trace- and method-based compilers. Details are explained in Section 4.2.1.

In addition to the leverage strength of the two JIT compilation strategies, we aim to resolve the path-divergence problem by selectively applying method-based compilation to the functions that cause the problem trace compilation to the other parts of a program. Since this proposal focuses on combining the different two strategies, dynamically selecting a suitable strategy depending on target programs' structure is left as future work.

Figure 4.1 gives an overview of our multi-scope meta-tracing JIT compiler framework. As shown in Figure 4.1a, when a language developer writes interpreter definition with the framework, the framework can generate a VM with a multi-scope JIT compiler. Figure 4.1b overviews our multi-scope compilation and runtime by our framework. At runtime, the generated multi-scope JIT compiler applies different strategies to different parts of a source program. Further, machine codes generated from different strategies can move back and forth with each other by *Stack Hybridization*, which is illustrated in Section 4.3.

4.2 Mixing The Two Compilation Strategies in Meta-Level

In this section, we first describe how to construct method-based compilation based on (meta-) trace-based compilation. We then explain how to cooperate with them in a meta JIT compiler framework. In this work, we merely aim at achieving *simple* method compilation; i.e., advanced optimization techniques used in existing method JIT compilers are left for future work.

4.2.1 Method-Based Compilation by Tracing

To construct method-based compilation by utilizing a trace-based compilation, we have to cover all paths of a function. In other words, we need to determine the *true path* and decrease the number of guard failures that occur to solve the path-divergence problem when applying trace-based compilation.

This dissertation proposes *method JIT by tracing* by customizing the following features of trace-based compilation: (1) trace entry/exit points, (2) conditional branches (3) loops, (4) function calls. In the following paragraphs, the author explain in detail how to "trace" a method by modifying these features. Note that method JIT by tracing is more naive than other state-of-the-art method-based JIT compilers, since this method-based compilation is designed for applying for programs with complex control flow, which causes

Algorithm 3: JitMetaMethod(rep, states, residue)

```
input  : Representations of an interpreter itself.
input  : States (e.g., virtual registers and memories) of an interpreter
         itself.
input  : An array data structure that records an executed instruction.
output: The trace of a target method in a source program
if op = method_entry then
    residue ← [ ];
    do
        if op = conditional branch then
            TraceCond (rep, states, residue);
        else if op = loop entry then
            TraceLoop (rep, states, residue);
        else if op = function call to f then
            TraceFunction (rep, states, residue);
        else
            eval(op, states, residue);
        op ← rep.get_next(op);
    while op != return;
    return residue;
else
    return;
```

performance degradation problem in a trace-based compilation, and we apply a trace-based compilation for other programs. Thus, the trace-based compiler is the primary compiler, and the method-based compiler is the secondary compiler in our system.

Trace entry/exit points Trace-based JIT compilers [Gal, Probst, and Franz, 2006; Gal et al., 2009] generally compile loops in the source program; therefore, they start to trace at the top of a loop and end when the execution returns to the entry point. To assemble the entire body of a function, we modify this behavior to trace from the top of a method body until a `return` instruction is reached (see Algorithm 3).

Conditional branches When handling a conditional branch, trace-based JIT compilers convert a conditional branch into a guard instruction and collects instructions that are executed. When execution method-based compilation, however, we must compile both sides of conditional branches. To achieve this, a tracer that records executed instructions must return to the branch point and restart tracing the other side as well. As shown in Algorithm 4, the tracer in our constructed method-based JIT compiler has to trace both then and else sides so that it *backtracks* to the beginning of a conditional branch when it reaches the end of one side and continues to trace the other side. Before starting to trace one side, the tracer stores its states (e.g., the data

Algorithm 4: TraceCond(rep, states, residue)

```
regs, mems ← [ ], [ ];
do
    regs.store(states.get_reg());
    mems.store(states.get_mem());
    trace_then ← JITMETAMETHOD(states);
    states.restore(regs, mems);
    trace_else ← JITMETAMETHOD(states);
    // construct if exp including trace_then and trace_else
    trace_ifexp ← begin
        if op.const_fold(states) then
            trace_then;
        else
            trace_else
    residue.append(trace_ifexp);
    op ← rep.next_of(op);
while op != return;
```

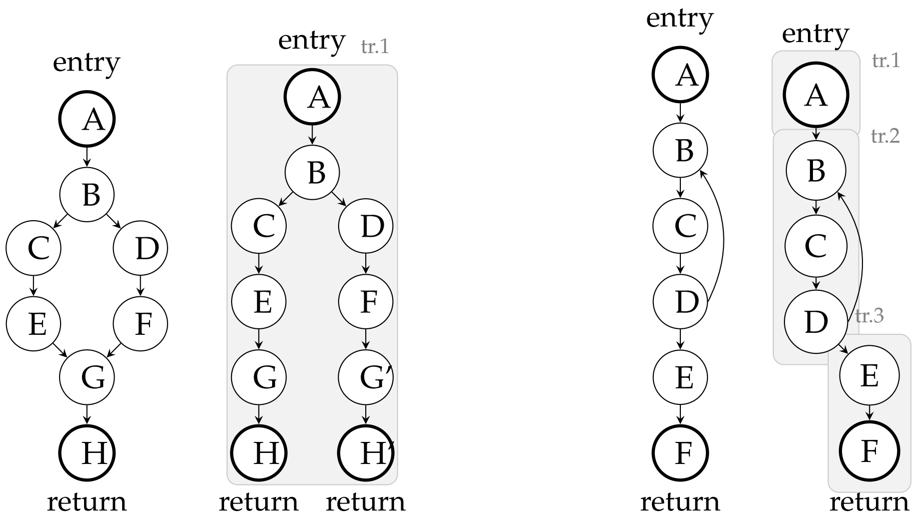
stored in the tracer's virtual registers and memories) in already prepared arrays. For just backtracking, the tracer *restores* those states and continues to the other side.

Figure 4.2a shows an example describing how the tracer for method-based compilation works. On the left side, node A is the method entry, nodes B – C – D form a conditional branch, and node E is the end of this method. The tracer starts to trace at A. On reaching a conditional branch (B), the tracer then stores its state and follows one side (B – C – E – G – H). On reaching *return* instruction (H), the tracer finally backtracks to B and resumes to trace the other side (B – D – F – G – H) by restoring the already saved data.

There is a risk of an exponential blow-up of compiled code when tracing a program that has many nested conditionals. To avoid generating too big native code, when the compiler detects too many branches in a target program part, our system stops the method-based compiler to trace. Instead, our system switches to apply trace-based compilation for such a program.

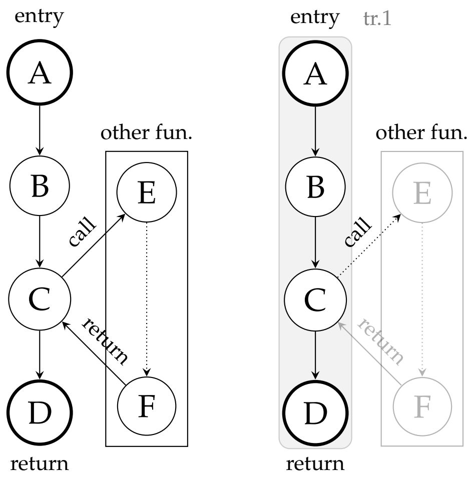
Loops Our method-based compiler does not handle loops specially. While a trace-based compiler compiles loops as a straight-line path, our method-based compiler compiles not only the body a target loop, but also the successors of it.

Algorithm 5 illustrates how the tracer for method-based compilation traces a loop. When the tracer finds the entry point, it starts to analyze the body of a function to find a back-edge and loop-exit instruction. When the tracer traces a back-edge, as with a trace-based compilation, leaves an instruction to `jump` to the entry. When the tracer leaves a loop-exit instruction, it also traces the


entry
A
B
C
D
E
F
G
H
return
entry
tr.1
A
B
C
D
E
F
G
G'
H
H'
return
return
entry
A
B
C
D
E
F
return
entry
tr.1
A
tr.2
B
C
D
tr.3
E
F
return

(A) Handling of a conditional branch. In this program, A represents a method entry, and B – C – D represents a conditional branch.

(B) Handling of a loop. In this program, B – C – D represent a loop, and E – F is a successor of the loop B – C – D.


entry
A
B
C
D
return
other fun.
E
F
call
return
entry
tr.1
A
B
C
D
return
other fun.
E
F
call
return

(C) Handling of a function call. A – B – C – D and E – F are functions. In this program, (A – B – C – D) calls (E – F) at C. Note that only target function (A – B – C –D) is compiled.

FIGURE 4.2: Examples how our method-based compilation works. Each left-hand side is the control-flow of a target source program that represents one method, and each right-hand side is a result. “entry” and “return” means the entry point and exit poitn of a target method, respectively.

destination of a loop-exit instruction and leaves a `jump` instruction to go to the outside of this loop.

Figure 4.2b shows an example of how to handle a loop. In this example, our method-based compiler compiles a single loop into three trace parts. The first one (tr.1) is up to the loop entry, the second one is the loop itself (tr.2) of the loop, and the third one is the successor of the loop (tr.3).

Algorithm 5: TraceLoop(rep, states, residue)

```
op ← rep.get_op();
do
    if op = back-edge to the entry then
        residue.append(jump to entry);
    else if op = loop-exit then
        loop_after_state ← rep.next_of(op).get_states();
        loop_after_trace ← JITMETAMETHOD(rep, loop_after_state);
        residue.append(jump to loop_after_trace);
        residue_loop_after.append(loop_after_trace);
    else
        eval(op, states, residue);
    op ← rep.next_of(op);
while op!= return;
residue.append(residue_loop_after)
```

Algorithm 6: TraceFunction(rep, states, residue)

```
do
    if op = function call to f then
        residue.append(call to f) ; // not following but leaving the
         instruction "call f"
        // continue to trace successors
while op != return;
```

Function calls Whereas a trace-based JIT compiler will inline function calls, our method-based JIT compiler will not inline, but emit a call instruction code and continue tracing. We don't inline function because our method-based compilation is designed to apply only for programs with the path-divergence problem. If a target program needs inlining, we will apply trace-based compilation for it, since trace-based compilation can automatically perform function inlining. Thus, our method-based compilation is so naive that it is not equivalent to other method-based JIT compilers.

To remain a function call in a resulting trace, we have to inform which part is represented to a source program function call in an interpreter definition. Therefore, we need to implement the specific interpreter style shown in the left-hand side of Figure 4.3 (we call this style *host-stack style* here). By writing in host-stack style, the tracer can detect which part is a source program's method invocation and leave a call instruction in a resulting trace. Figure 4.2c shows how the tracer compiles a function call. In this example, the tracer eventually generates one trace, including a call instruction (tr.1).

In trace-based compilation, however, a meta-tracing JIT compiler can work efficiently in a specific way as shown in the right-hand side of Figure 4.3 (we call this *user-stack style* here).

```
if opcode == CALL:
  addr = self.bytecode[pc]
  # call the `interp'
  # recursively
  res = self.interp(addr)
  user_stack.push(res)
  pc += 1
elif opcode == RETURN:
 # return a top of
 # `user-stack'
  return user_stack.pop()
```

(A) Example interpreter written in host-stack style.

```
if opcode == CALL:
  addr = self.bytecode[pc]
  pc += 1
  # push a return addr to
  # `user-stack'
  ret_addr = W_IntObject(pc)
  user_stack.push(ret_addr)
  if addr < pc:
    can_enter_jit(..)
  # jump to a callee function
  pc = t
elif opcode == RETURN:
  v = user_stack.pop()
  # restore already pushed
  # return addr
  addr = user_stack.pop()
  user_stack.push(v)
  if addr < pc:
    can_enter_jit(..)
  # jump back to the caller
  # function
  pc = addr
```

(B) Example interpreter written in user-stack style.

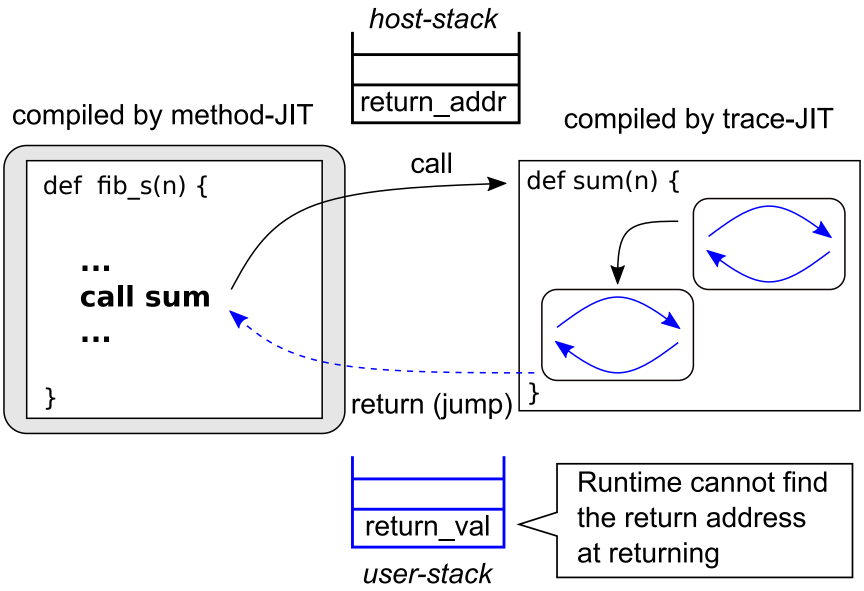
FIGURE 4.3: Interpreter definition styles. For managing a return address/value, left-hand side style uses a host-language's (system provided) stack, but right-hand side uses a developer-prepared stack data structurpe.

In the next section, we organize why we need the two different two stack styles.

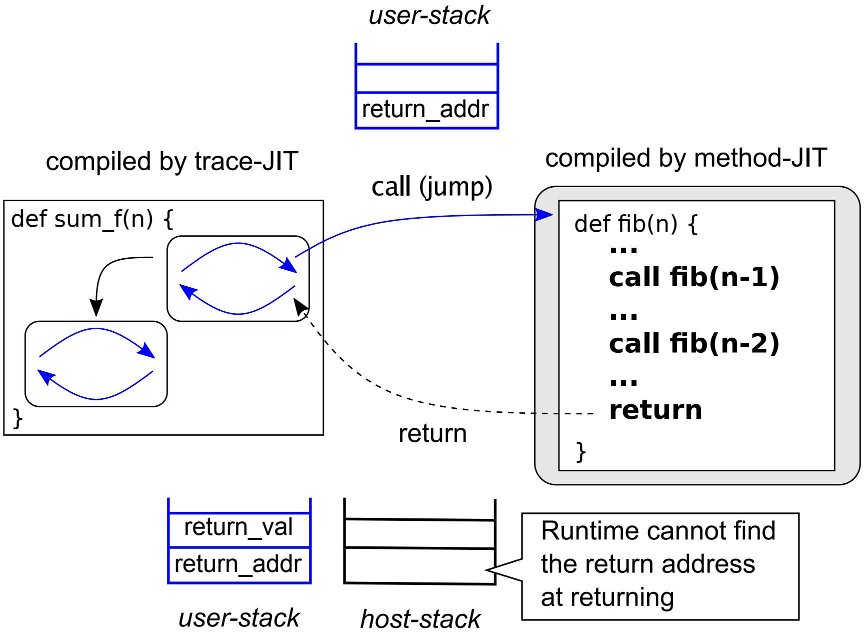
4.3 Stack Hybridization

4.3.1 Combination Problem

The reason why the author cannot naively combine them is the following: the two compilations require different interpreter implementation styles in function calls. Trace-based compilation requires the *user-stack style*, while method-based compilation requires the *host-stack style*. In other words, different types of compilations use different stack frames for optimizing function calls. Because of this gap, the runtime cannot call back and forth between native codes generated from the two compilations. Trace-based compilation inlines a function call; therefore, there is no function call instruction in the resulting trace. Whereas method-based compilation "leaves" a function call instruction in the resulting trace. We explain this problem by using Figure 4.4. Figure 4.4a shows an example that a method-compiled function calls a trace-compiled function, and Figure 4.4b shows an example that a method-compiled function calls a trace-compiled function.


host-stack
return_addr
compiled by method-JIT
compiled by trace-JIT
call
def fib_s(n) {
...
call sum
...
}
def sum(n) {
}
return (jump)
return_val
user-stack
Runtime cannot find the return address at returning

(A) Calling a trace-compiled code from a method-compiled function.


user-stack
return_addr
compiled by trace-JIT
compiled by method-JIT
call (jump)
def sum_f(n) {
}
def fib(n) {
...
call fib(n-1)
...
call fib(n-2)
...
return
}
return
return_val
return_addr
user-stack
host-stack
Runtime cannot find the return address at returning

(B) Calling a method-compiled function from a trace-compiled code.

FIGURE 4.4: Example of Combination Problem. Gray background code is compiled by method JIT, and blue lined code is compiled by tracing JIT.

In the case that `fib_s` (compiled by method-based compilation) calls `sum` (compiled by trace-based compilation) as shown in Figure 4.4, the runtime puts a return address in the host-stack. In `sum`, the return value and return address are stored in the user-stack. On returning from `sum`, since the semantics of return is defined as shown in Figure 4.3, the runtime attempts to find a return address from a user-stack. However, the return address is stored in a host-stack, and the runtime cannot return to the correct place.

In the case shown in Figure 4.4b, `sum_f` (compiled by trace-based compilation) calls `fib` (compiled by method-based compilation), however the runtime puts its return address in the user-stack. When runtime returns from `fib`, it then attempts to find the return address from the host-stack, but it fails to find the address and results in runtime-error because the return address is pushed to the user-stack.

4.3.2 Bridging Native Code with Different Calling Conventions

To overcome this problem, the author proposes *Stack Hybridization*, a mechanism to bridge the native code generated from different strategies. Stack Hybridization manages different kinds of stack frames, and generates native code that can be mutually executed in trace-JIT and method-JIT contexts. To use Stack Hybridization, a language developer needs to write an interpreter in the specific way: (1) For executing a call instruction in the base language, developers put a special flag to indicate which stack frame is used in a self-prepared stack data structure. (2) For executing a return, they have to branch to the return instruction of the base language corresponding to the call by checking the already pushed flag.

Roughly speaking, the interpreter handles the call and return operations in the following ways:

When it calls a function under the trace-based compilation, it uses the user-stack; i.e., it saves the context information in the stack data structure, and iterates the interpreter loop. Additionally, it leaves a flag “user-stack” in the user-stack.

When it calls a function under the method-based compilation, it uses the host-stack; i.e., it calls the interpreter function in the host language. Additionally, it leaves a flag “host-stack” in the user-stack.

When it returns from a function, it first checks a flag in the user-stack. If the flag is “user-stack”, it restores the context information from the user-stack. Otherwise, it returns from the interpreter function using the host-stack.

To support behaviors, the author introduces an interpreter implementation style, which enables to embed both styles into a single interpreter and switch its behavior depending on the flag. Figure 4.5 shows a sketch of special syntax to support Stack Hybridization. The important syntaxes are `is_mj` pseudo function, and `US`/`HS` special flags. `US` means a flag “user-stack”, and `HS` means a flag “host-stack”.

`is_mj` is used for selecting suitable `CALL` definitions at compilation time. This pseudo function returns `true` under method-based compilation context, otherwise `false`. The host-stack styled definition should be placed in the `then` branch, and the user-stack styled definition is placed in the `else` branch as

```
if instr == CALL:
 addr = bytecode[pc]
 # branch considering by
 # a JIT ctx.
 if is_mj():
  # push JIT flag (HS) to
  # ``user-stack''
  user_stack.push(HS)
  ret_val = interp(addr)
  user_stack.push(ret_value)
 else:
  # push JIT flag (US) to
  # ``user-stack''
  user_stack.push(US)
  user_stack.push(pc+1)
  pc = addr
```

```
elif instr == RETURN:
 ret_val = user_stack.pop()
 # get JIT ctx. flag from
 # ``user-stack''
 JIT_flg = user_stack.pop()
 # check the JIT ctx. and branch
 if JIT_flg == HS:
  return ret_val
 else:
  ret_addr = user_stack.pop()
  user_stack.push(ret_val)
  pc = ret_addr
```

FIGURE 4.5: A sketch of a interpreter definition with Stack Hybridization. Some hint functions (e.g., `can_enter_jit` and `jit_merge_point`), and other definitions are omitted for simplicity. `US` and `HS` represents user-stack and host-stack, respectively.

shown in the left of Figure 4.5. Then, the multi-scope meta-tracing JIT compiler traces the `then` branch in the context of trace-based compilation, but traces the `else` branch under method-based compilation context.

`US` and `HS` mean trace- and method-based compilation contexts, respectively. These special variables are used for detecting JIT compilation context dynamically when executing `RETURN` at runtime (not compilation time).

When defining `CALL` in an interpreter, `US` or `HS` is placed at the top of a user-stack when language developers define `CALL` instruction. At compilation time, these flags are treated as *red* variable, so an instruction pushing `US` or `HS` flag is left in a resulting trace. The compiler also leaves the branching instruction (`if JIT_flg == HS: ... else: ...`) in a resulting trace when tracing `RET`. This enables to find a JIT compilation context, and cooperate resulting traces made from the different two strategies at runtime.

For example, there are two traces, one (A) is made from trace-based compilation and the other (B) is from method-based compilation. When a function call from A to B is occurred, a flag `US` is pushed to a user-defined stack. When executing a `RET` instruction in B, the control executes a suitable definition by writing as shown in the right of Figure 4.5.

4.4 Evaluation

The author implemented the BacCaml multi-scope meta-tracing JIT compiler framework based on the MinCaml compiler [Sumii, 2005]. MinCaml is a small ML compiler designed for education-purpose. MinCaml can generate native code almost as fast as other notable compilers such as GCC or OCamlOpt. The author did not extend RPython itself because the implementation of RPython is too huge to comprehend. As an initial step, the author created a subset of RPython on a compiler with reasonable implementation size [1].

4.4.1 Setup

To assess the performance of our BacCaml framework, we executed two kinds of benchmarks. The first one, namely standalone JIT microbenchmark, is for evaluating the performance of BacCaml's trace-based compilation and method-based compilation, respectively. The second one, namely multi-scope JIT microbenchmark, is for evaluationg the quality of our multi-scope JIT compilation strategy.

Target language: MinCaml−− The author also created a small functional programming language, namely MinCaml−−, with the BacCaml framework for taking microbenchmark. It is almost same to MinCaml, but limited to unit, boolean and integer variables [2].

Methodology

The author attempted to run all of MinCaml's test programs [3] and shootout [4] benchmark suite by MinCaml−− and BacCaml before taking microbenchmark. Then, we selected all programs that can be successfully worked in them. The names of microbenchmark programs are shown in the X-axis of Figure 4.6.

When taking microbenchmark, a threshold for starting JIT compilation is set at a lower-than-normal value to simplify the situation. Basically, it sets 100 as a threshold to determine whether starting JIT compilation or not. Therefore, this microbenchmark can arrive at a steady-state by attempting at most 50 iterations. Thus, it ran each program 150 times, and the first 50 trials were ignored to exclude the warm-up.

Since BacCaml is a prototype, it converts a resulting trace to Assembly by using by GCC at the compilation phase. The compiler then dispatches the control to the machine code by using dynamic loading in the execution phase. Particularly, trace-generation and compilation processes consume much time (approximately 80 % of a warm-up phase), so using a JIT native code generation framework such as libgccjit [5] or GNU Lightning [6] is left as future work.

---

[1] BacCaml itself is written in OCaml, and its implementation can be accessed at GitHub (`https://github.com/prg-titech/BacCaml`)

[2] MinCaml−− is also available at GitHub (`https://github.com/prg-titech/MinCaml`)

[3] `https://github.com/esumii/min-caml/tree/master/test`

[4] `https://dada.perl.it/shootout/`

[5] `https://gcc.gnu.org/onlinedocs/jit/`

[6] `https://www.gnu.org/software/lightning/`

It ran all the microbenchmarks on Manjaro Linux with Linux kernel version 5.6.16-1-MANJARO and dedicated hardware with the following modules; CPU: AMD Ryzen 5 3600 6-Core Processor; Memory: 32 GB DDR4 2666Mhz PC4-21300.

In the Figure 4.6a and Figure 4.6b, ⬚ means BacCaml's tracing JIT, ▮ means BacCaml's method JIT, ▩ means BacCaml's interpreter-only execution, □ means MinCaml (AOT), and ✶ means BacCaml's multi-scope JIT (mixing tracing and method JIT).

Threats to Validity

There are the following threats to validity in our evaluation (including the experiment in the next section). The method-based compilation in BacCaml is naive, then the inference which trace-based compilation is faster than method-based compilation has a possibility to be overturned by a full-fledged method-based compilers. This is actually true not only to our method-based compiler, but also for our trace-based compiler when compared against the state-of-the-art trace-based compilers like PyPy.

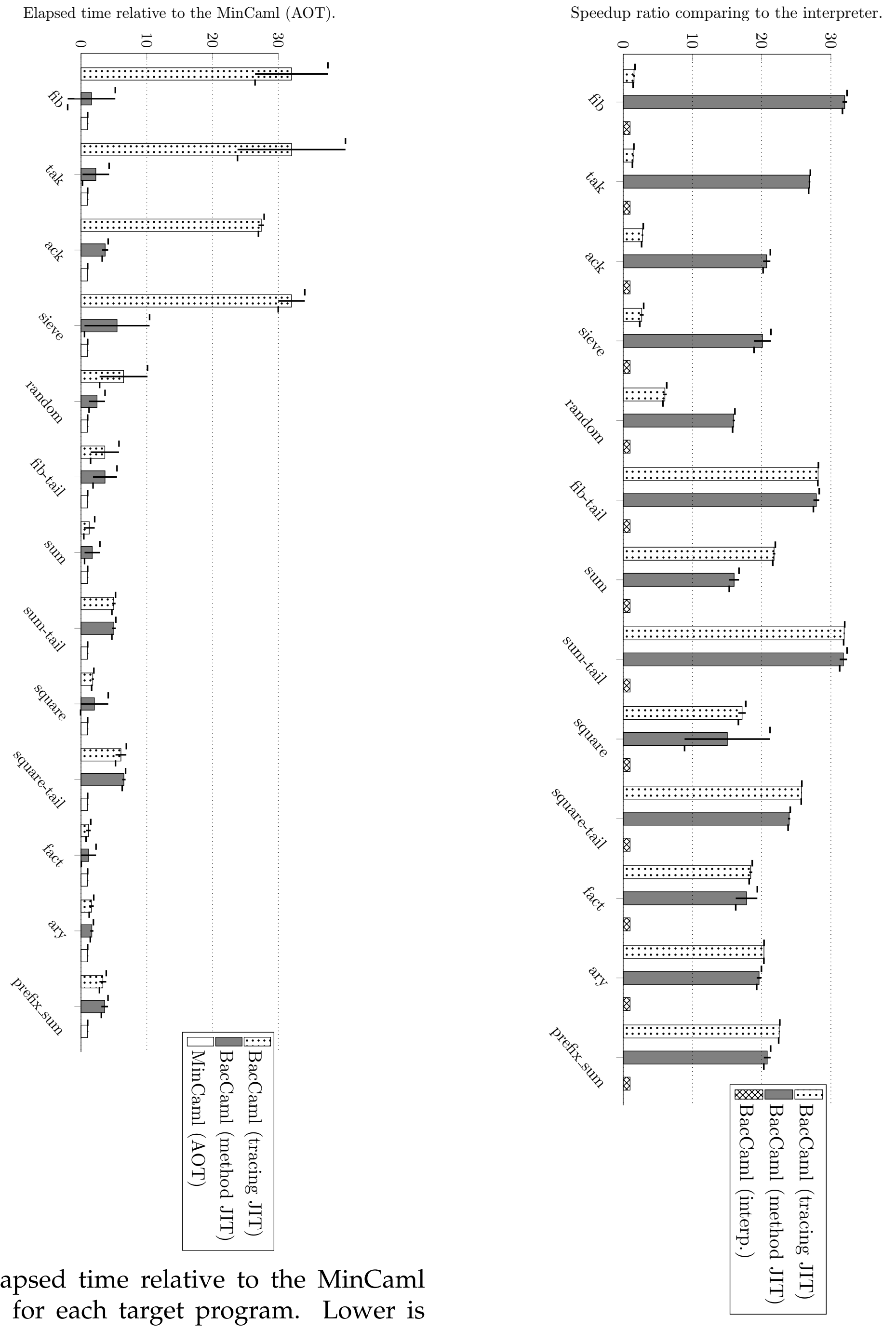
4.4.2 Standalone JIT Microbenchmark

For comparing the standalone performance of trace- and method-based compilation strategies, the author first applied both strategies separately for programs written in MinCaml−−, and compared the performances of MinCaml−− with JIT with an interpreter-only execution of MinCaml−− and the MinCaml ahead-of-time compiler.

Before showing data, it explains the limitations of our method-based compilation. Compared to other state-of-the-art method JIT compilers, BacCaml's method-based compilation does not inline them. It is because our method-based compilation is designed to be applied to only programs with the path-divergence problem. In other words, it is a fallback strategy when trace-based compilation does not work well.

The results are shown in Figure 4.6a and 4.6b. Note that Figure 4.6a is normalized to the MinCaml (lower is better), but Figure 4.6b is to the interpreter-only execution (higher is better).

Figure 4.6a illustrates the performances of the two JIT compilations comparing to the elapsed time of MinCaml−− (AOT). In these results, our trace-based compilation (⬚) was from 1.12 to 12.4x slower than MinCaml (AOT) (□) in programs which have straight-forwarded control flow (fib-tail, sum, sum-tail, square, square-tail, fact, ary, prefix_sum). BacCaml's trace-based compilation was effective on such programs since almost all executions are run on compiled straight-line traces. However, it performs from 38.5 to 42.1x slower than other strategies in programs with complex control flow (fib, ack, tak, sieve), since these programs cause the path-divergence problem. In Figure 4.6b, BacCaml's trace-based compilation was from 6.02 to 31.92x faster



(A) Elapsed time relative to the MinCaml (AOT) for each target program. Lower is better.

(B) Speedup ratio of JITs comparing to the interpreter-only execution for each target program. Higher is better.

FIGURE 4.6: Results of standalone JIT microbenchmarking. The five programs on the left have a complex control flow, and the remaining programs have a straight control flow. The error bars represent the standard deviations.

than interpreter-only in programs with straight-line control flow. However, it was still from 1.44 to 2.68x faster than interpreter-only in programs with the path-divergence problem, since most of the execution was done on the interpreter.

On the other hand, our method-based compilation (▮) was from 1.16 to 6.52x slower than MinCaml (AOT) in Figure 4.6a. Besides, from Figure 4.6b, our method-based compilation also performs from 15.04 to 37.8x faster than interpreter-only. From these results, most execution ran on a resulting trace. Our strategy prevented the path-divergence problem since our method-based compilation covered the entire body of a method.

Overall, our trace-based compilation was about 1.10x faster in programs with straight-line control flow but about 11.8x slower in programs with complex control flow than our method-based compilation. According to those results, we can say that trace-based compilation's performance depends on the control flow of a target program (when fitted to trace-based strategy, its performance was better than method-based strategy). Still, a method-based strategy works well on average. Therefore, we argue that combining the two strategies is vital for further speedup on JIT compilation.

4.4.3 Multi-Scope JIT Experiment

In this section, we demonstrate the result of an experiment for a multi-scope JIT compilation strategy. This experiment aims to confirm if there are programs that are faster with a multi-scope strategy than standalone strategies. This experiment was also performed by the same implementations used in Section 4.4.2.

Methodology

We first synthesized two types of functions, one is suitable for trace-based compilation, and the other is for method-based compilation according to the result shown in Section 4.4.2. Then, we applied multi-scope JIT compilation for them; trace-based compilation is applied for a program with straight-line control flow, and method-based compilation is applied for a program with complex control flow. Finally, we compared the performance with standalone JIT strategies (tracing JIT only and method JIT only) and BacCaml's interpreter-only execution.

According to the result shown in Section 4.4.2, we chose sum, fib, and tak from the microbenchmark programs. It is because sum is faster in trace-based compilation than in method-based compilation, and fib and tak are faster in method-based compilation than a trace-based compilation. Then we manually synthesized those functions for preparing test programs, namely sum-fib, fib-sum, sum-tak and tak-sum, that shown in Figure 4.7.

For taking data, we used the same implementations and hardware employed in Section 4.4.2. We also took 150 iterations and ignored the first 50 trials to exclude warm-up.

```
let rec fib n =
  if n <= 1 then 1 else
    fib (n-1) + fib (n-2) in

let rec sum i n =
  if i <= 1 then n else
  let m = fib 10 in
    sum (i-1) (n+m) in

print_int (sum 1000 0)
```

(A) sum-fib

```
let rec sum acc n =
  if n <= 1 then acc else
    sum (acc+n) (n-1) in

let rec fib n =
  if n <= 2  then sum 0 1000
  else
    fib (n-1) + fib (n-2) in

print_int (fib 20)
```

(B) fib-sum

```
let rec tak x y z =
  if x <= y then z else
    tak (tak (x-1) y z)
        (tak (y-1) z x)
        (tak (z-1) x y) in

let rec sum i n =
  if i <= 1 then n else
  let m = tak 12 6 4 in
    sum (i-1) (n + m) in

print_int (sum 100 0)
```

(C) sum-tak

```
let rec sum i n =
  if i <= 1 then n else
    sum (i-1) (n+i) in

let rec tak x y z =
  if x <= y then sum 1000 0
  else
    tak (tak (x-1) y z)
        (tak (y-1) z x)
        (tak (z-1) x y) in

print_int (tak 8 4 2)
```

(D) tak-sum

FIGURE 4.7: Target programs written in MinCaml−− used for the multi-scope JIT experiment.

Since the algorithm that decides to apply which compilation strategy to which part of a program is left for future work, we manually decided the program parts' strategies. In a multi-scope JIT compilation strategy, we applied trace-based compilation to sum, and method-based compilation to fib and tak manually. Despite this, in other strategies, we used only a single strategy for those test programs. Specifically, we applied trace-based compilation to sum, fib and tak in a tracing JIT only strategy, and applied method-based compilation for them in a method JIT only strategy.

Results of Multi-Scope JIT Experiment

The results of the multi-scope JIT experiment are shown in Figure 4.8. Overall, our multi-scope compilation strategy (◧) was from 1.01 to 2.17x faster than our method-based compilation-only. Our multi-scope JIT strategy avoided the overhead of recursive function calls in sum when executing the native code generated from sum, since the recursive call part was inlined by trace-based compilation.

In contrast, our multi-scope strategy was from 1.01 to 1.59x faster than the trace-based compilation-only in fib-sum and tak-sum. Moreover, our multi-scope strategy was about 20x faster in sum-fib and sum-tak. This difference was caused by the structure of the target program's control flow. In fib-sum and tak-sum, fib and tak can be connected to the sum's recursive call and return parts. Otherwise, in sum-fib and sum-tak, sum cannot be connected

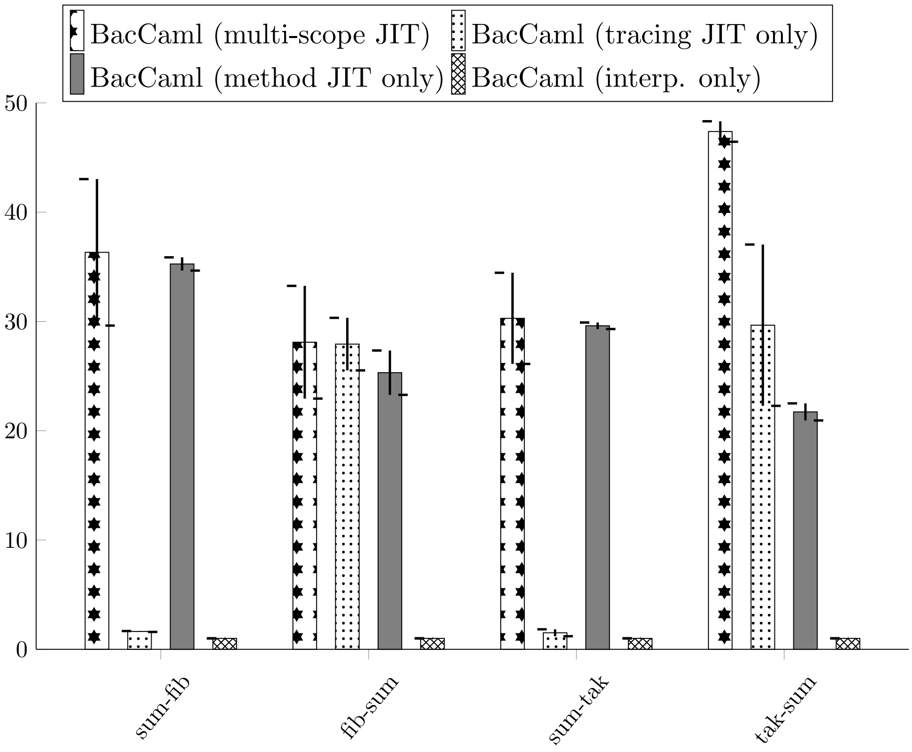


FIGURE 4.8: Results of multi-scope JIT microbenchmarking. X-axis represents the name of a target program, and Y-axis represents speedup ratio relative to the interpreter-only execution. Higher is better. The error bars represent the standard deviations.

to entire fib and tak, since our trace-based compiler cannot cover entire body of fib and tak functions by the path-divergence problem.

From the results, we can report that there are programs that can be run faster by the multi-scope compilation strategy.

4.5 Related Work

Self-optimizing Abstract-syntax-tree interpreter. Self-optimizing abstract-syntax-tree interpreter [Würthinger et al., 2012] also enables language developers to implement effective virtual machines. The framework and the compiler are called Truffle and Graal, respectively. The difference from our system is the basic compilation unit. Our system is based on a meta-tracing compiler, so the compilation unit is a trace. In contrust, Truffle/Graal applies partial evaluation for an AST-based interpreter of an interpreter at execution time. By profiling the runtime types and values, it can optimize a source program and run it efficiently.

GraalSqueak. GraalSqueak [Niephaus, Felgentreff, and Hirschfeld, 2018] is a Squeak/Smalltalk VM implementation written in Truffle framework. In [Niephaus, Felgentreff, and Hirschfeld, 2018], Niephaus et al. provided an efficient way to compile a bytecode-formatted program; that is, they showed a way to apply trace-based compilation with an AST-rewriting interpreter strategy.

We extend the meta-tracing JIT compilation framework to support method-based compilation, but their approach involves creating an interpreter to enable trace-based compilation on a partial evaluation-based meta-JIT compiler framework. Their idea is to implement an interpreter with some specific hint annotations to expand the loop of an application program. Sulong has already demonstrated the same idea [Rigger et al., 2016], and it was applied for implementing Squeak/Smalltalk VM.

Region-based JIT compiler. HHVM [Ottoni, 2018] is a high-performance VM for PHP and Hack programming languages. An important aspect of HHVM 2nd generation is its region-based JIT compiler. A region-based compiler [Hank, Hwu, and Rau, 1995] is not restricted to compile the entire body of methods, basic blocks, or straight-line traces; it can compile a combination of several program areas. Their compilation strategy is more flexible than our multi-scope compilation strategy, because an HHVM region-based compiler can compile basic blocks, the entire body of methods, loops, and any combination of them. However, their approach is limited to a specific language system; we aim to provide some flexibility of compilation as a meta JIT compiler framework.

Lazy basic block versioning. Lazy basic block versioning [Chevalier-Boisvert and Feeley, 2015, 2016] is a JIT compilation technique based on basic blocks. This strategy combines type specialization and code duplication to remove redundant type checking, and it can generate effective machine code. Moreover, as well as HHVM's region-based JIT compiler, it can compile straight-line code paths and the entire method bodies by constructing basic blocks. Chevalier-Boisvert and Feely implemented lazy basic block versioning on Higgs, a research-oriented JavaScript VM [7]. Their method-based compilation is similar to ours; tracing both sides of conditional branches, not inlining functions, and not handle loops specially. The difference is that our method-based compilation is not based on basic blocks, but on traces. Moreover, our multi-scope compilation can be applied not only for specific, but also for any languages.

HPS: High Performance Smalltalk. High Performance Smalltalk (HPS) is a virtual machine used in VisualWorks Smalltalk [Miranda, 1999]. HPS achieves efficient performance by well-planned stack frame management. In HPS, the key technique for efficient implementation of contexts is mapping (closure or method) activations to stack frames in runtime. HPS has three context representaions. (1) *Volatile* contexts: precedure activations which have yet to be accessed as context objects. (2) *Stable* contexts: the normal object form of procedures. (3) *Multi-Scope* contexts: a pair of a context object and its associated activation. By preparing extra slots for multi-scope contexts in the stack frames, HPS can distinguish between multi-scope and volatile contexts. This technique is similar to our stack hybridization. Stack hybridization also has the two contexts, which represent tracing and method JIT, respectively. Moreover, the stack hybridization checks the return pc by a flag in an user-defined array structure as well as HPS manages the context in a separate array object. To avoid impacting the garbage collector, the header of a multi-scope context is spotted as an object including raw bits rather than object pointers. However, stack hybridization is so naive that it currently does not consider the impact of garbage collector.

---

[7] `https://github.com/higgsjs/Higgs`

4.6 Conclusion

This chapter proposed a multi-scope meta-tracing JIT compiler framework to take advantage of trace- and method-based compilation strategies as a multilingual approach. For supporting the idea, it chose a meta-tracing JIT compiler as the foundation and extended it to perform method-based compilation using tracing. By customizing the following features, it realized it: (1) trace entry/exit points, (2) conditional branches, (3) function calls, and (4) loops. The chapter also proposed Stack Hybridization: an interpreter design to enable connecting native code generated from different strategies. The key concept of Stack Hybridization is (1) embedding two types of interpreter implementation styles into a single definition, (2) selecting an appropriate style at just-in-time compilation time, and (3) putting a flag on the stack data structure to indicate whether it is under trace- or method-based compilation.

For implementation, the chapter proposes a prototype of a multi-scope meta-tracing JIT compiler framework called BacCaml as a proof-of-concept. It created a small meta-tracing JIT compiler on the MinCaml compiler, and supported method-based compilation by extending trace-based compilation, and achieved Stack Hybridization on it.

Through the evaluation the basic performance of BacCaml's trace- and method-based compilers, the results showed that our trace-based compiler ran from 6.02 to 31.92x faster than interpreter-only execution in programs with straight-line control flow, but 1.44 to 2.68x faster in programs with complex control flow. Its method-based compiler ran 15.04 to 37.8x faster than interpreter-only execution in all types of programs. It finally executed a synthetic experiment to confirm the usefulness of the multi-scope strategy, and reported that there are example programs that are faster with the multi-scope strategy.

Chapter 5

Threaded Code Generation with a Real-World Meta-Tracing JIT Compiler

Contents

- **5.1 Introduction . . . 60**
- **5.2 Threaded Code Generation . . . 60**
  - 5.2.1 Threaded Code . . . 61
  - 5.2.2 The Compilation Principle . . . 61
  - 5.2.3 Method-traversal Interpreter . . . 64
  - 5.2.4 Trace Stitching . . . 67
  - 5.2.5 Shallow Tracing . . . 69
- **5.3 Runtime Techniques for Multilevel Compilation . . . 70**
  - 5.3.1 Implementaiton Details . . . 71
- **5.4 Optimization for Threaded Code Generation with Interpreter in the Meta-Tracing JIT Compiler . . . 72**
  - 5.4.1 Inline Caching in Method-Traversal Interpreter . . . 72
- **5.5 Preliminary Evaluation Using Simulated Threaded Code Generation . . . 74**
  - 5.5.1 Simulated Threaded Code Generation (STCG) in PyPy 75
  - 5.5.2 Setup . . . 76
  - 5.5.3 Results of Experiment 1: The Overhead of Our STCG 77
  - 5.5.4 Results of Experiment 2: The Stable Speed . . . 78
  - 5.5.5 Discussion . . . 78
- **5.6 Evaluation and Experiments in PySOM and Multilevel RPython . . . 79**
  - 5.6.1 Microbenchmark Evaluation . . . 79
  - 5.6.2 Multilevel JIT Experiment . . . 83
- **5.7 Related work . . . 85**
  - 5.7.1 Improving an Interpreter's Performance . . . 85

5.7.2 Template JIT Compilation 85
5.7.3 Ahead-of-Time Compilation 86
5.7.4 Introducing a New Behavior into a Meta-JIT Compiler 86
5.8 Conclusion 86

5.1 Introduction

This chapter proposes *Multilevel RPython*, which supports multilevel JIT compilation, combining method-based lightweight JIT compilation and trace-based heavyweight JIT compilation in a real-world meta-tracing JIT compiler.

Multilevel RPython can use the two compilation policies: method-based lightweight and trace-based heavyweight compilations. The overview of Multilevel RPython is shown in Figure 5.1. First, the user of Multilevel RPython writes an interpreter and inserts hint instructions into it. Then, Multilevel RPython generates a VM using the multilevel JIT compiler, and the written interpreter is divided into two interpreters: one is for method-based lightweight compilation and the other is for trace-based heavyweight compilation. The lightweight compilation is based on *threaded code generation* [Izawa et al., 2022], by taming an interpreter with hint hint instructions. The techniques for lightweight compilation is described in Sections 5.2.3, 5.2.4, and 5.2.5.

To switch the compilation level, Multilevel RPython adopts a profiling-based strategy. At runtime, the two are connected by using RPython's exception as global jumps, allowing for two different compilation policies to be used at runtime. This *interpreter shifting* technique is illustrated in Section 5.3.

5.2 Threaded Code Generation

This section presents how to realize method-based lightweight JIT policy on top of RPython. It is realized not by creating different compilers from scratch, but by allowing meta-tracing JIT compilation to behave differently with the use of taming an interpreter that is given to RPython.

The purpose of introducing threaded code generation is to improve the overall performance of the application by using it in conjunction with a tracing JIT in Multilevel RPython[1]. The hot spots are handled by the tracing JIT, which can generate high-quality code, even though this takes compile time, and the other warm spots are handled by threaded code generation, which can generate moderate quality code quickly. In general, a tracing JIT compiler automatically inlines function calls and applies several optimizations to a trace. The longer the trace and better native code intended for generation, the longer the compilation time will need to be. In contrast, threaded code generation only leaves the call instruction to a subroutine, so tracing does not consume much time. It is possible to balance code quality and compilation speed when the two policies have different optimization levels and different compilation units.

---

[1] An alternative approach to improve warm-up performance is to improve the dispatching mechanism of an interpreter, e.g., by using threaded jumps. It would not be easy to realize such approaches in RPython, because it currently assumes more straightforwardly written interpreters

5.2.1 Threaded Code

Before showing the compilation principle of threaded code generation, this section gives a brief overview of the fundamental technique called threaded code.

Threaded code [Bell, 1973; Hong, 1992] is a technique to improve the performance of a bytecode interpreter. The interpreter separately defines *handler functions* for all bytecode instructions, as shown on the right-hand side of Figure 5.2. A program is a sequence of call instructions to handlers, as shown on the left-hand side of the figure. Executing a threaded code-based program reduces the number of indirect branching that significantly pose a performance penalty at runtime because of branch mispredictions [Ertl and Gregg, 2003].

5.2.2 The Compilation Principle

Threaded code generation is achieved by taming an interpreter given to Multilevel RPython, which is instrumented to control the RPython's meta-tracing compiler and reconstructing the control flow from the resulted trace. Here, the *method-traversal interpreter* and *trace stitching* techniques achieve these. In the following, the approach is explained by comparing with a typical JIT compilation process in RPython.

When Multilevel RPython compiles a source program executed by an interpreter, it

starts compiling at the beginning of a loop, which is dynamically detected:

for each operation in the base program, it follows into the respective handler body in the interpreter, which effectively eliminates "interpretation" (i.e., code dispatching and operand manipulation) by the interpreter:

at a function call in the base program, it follows into the body of the callee function, which effectively achieves function inlining:

at a conditional expression in the base program, it follows only one of the branch by emitting a *guard failure*[2] for the other branches: and

---

[2] A guard failure is a runtime check to ensure the correctness of the generated trace.

at a conditional branch in the handler (including selection of an arithmetic operation based on operands' runtime types), it traces only one of the branch to achieve effective type specialization:

finishes compiling at the end of the loop.

Threaded code generation control the RPython compiler so that it

starts compiling at the beginning of a method/function in the source-program[3]:

for each operation in the base-program, it follows the code dispatching part of the interpreter, but it does not trace into the handler body but emits a call instruction to the respective handler;

at a function call in the base-program, emits a call instruction and continues tracing of the operations after the functional call;

at a conditional expression in the base-program, follows *all* branches;

at a conditional branch in the handler, this will not happen since the compiler does not trace the inside of handlers: and

finishes compiling at the end of the method/function.

To control the RPython's meta-tracing JIT compiler like that, the proposal consists of the following three techniques:

The *method-traversal interpreter* technique. Write an interpreter to let the tracing mechanism of RPython traverse all execution paths in a source program's method/function. This is achieved by inserting hint instructions into an interpreter.

The *trace stitching* technique. Reconstruct the original control flow of a base-program function/method from a recorded trace. Since the method-traversal interpreter technique will yield a straight-line trace that covers all the execution paths, this technique will split the trace into basic blocks and then connect them together by using branch and jump instructions. This is achieved by adding a post-processing module to the RPython tracer.

The *shallow tracing* technique. Naively applying a method-traversal interpreter to a meta-tracing compiler causes runtime inconsistency. A (meta-)tracing compiler executes the source program during tracing. Because threaded code generation traces paths that are not the actual execution path, it may make the interpreter's state (like operand stack) inconsistent, which will result in incorrect compiled code. To avoid this problem, this technique allows the tracer to follow the code without also executing it.

---

[3] Whether the system uses threaded code generation or not is an open issue that the author will consider in the future. For the time being, we merely assume that the threaded code generator is invoked for a particular base-program method/function.

```
@dont_look_inside
def tla_ADD(self, pc):
  x, y = self.pop(), self.pop()
  self.push(y.add(x))
  return pc

@dont_look_inside
def tla_CONST_INT(self, pc):
  arg = ord(self.bytecode[pc])
  self.push(W_IntObject(int(arg)))
  return pc + 1

threaded_driver = JitDriver(
  threaded_code_gen=True,
  reds=['self'],
  greens=['pc','bytecode','traverse_stack'])

def interp(self, pc, traverse_stack):
  threaded_driver.can_enter_jit(
      bytecode=self.bytecode,pc=pc,self=self,
      traverse_stack=traverse_stack)
  while True:
    threaded_driver.jit_merge_point(
      bytecode=self.bytecode,pc=pc,self=self,
      traverse_stack=traverse_stack)
    opcode = ord(self.bytecode[pc])
    pc += 1
    if opcode == ADD:
      pc = self.tla_ADD(pc)
    elif opcode == JUMP:
      ...
    elif opcode == RET:
      ...
    elif opcode == JUMP_IF:
      ...
```

LISTING 3: Skeleton of method-traversal interpreter and subroutines decorated with `dont_look_inside`.

Figure 5.3 shows a high-level example of threaded code generation. The left-hand side of Figure 5.3 represents the control flow of a target function. B – C – E is a conditional branch, D is a back-edge instruction, and F is a return. The compiler finally generates a trace tree [4], which covers a function body as shown on the right-hand side of Figure 5.3. In contrast to trace-based compilation, it keeps the original control flow so that it leaves the call instruction to a function are left in the resulting trace.

To produce such a trace tree keeping the original control flow, the tracer must sew and stitch the generated traces. Technically speaking, the compiler traces a special instrumented interpreter called the *method-traversal interpreter*. Because the trace obtained from the method-traversal interpreter ignores the

---

[4] Each trace has a linear control flow, but they are compiled as a bridge.

```
DUP,
CONST_INT, 1,
GT,
JUMP_IF, 10,
CONST_INT, 1
SUB,
JUMP, 0
CALL, 23,
EXIT,
```

LISTING 4: An example bytecode with the control flow shown in Figure 5.4.

```
while True:
    if x > 1:
        x -= 1
    else:
        x = call g(x)
        return x
```

LISTING 5: An example program corresponding to Listing 4

original control flow, it needs to restore it. To rebuild the original control flow, in the next phase, the lightweight JIT compiler stitches the generated trace. This technique is called *trace stitching*. The following sections explain the method-traversal interpreter and trace stitching, respectively.

5.2.3 Method-traversal Interpreter

The method-traversal interpreter allows a meta-tracing JIT compiler to behave in ways other than trace-based heavyweight compilation. In threaded code generation, the method-traversal interpreter allows it to emit method-based threaded code.

The skeleton of the method-traversal interpreter is shown in Listing 3. All handlers that are shown at the top of the listing, are decorated with the `dont_look_inside` hint instruction that tells the tracer not to trace the function body. Furthermore, specific areas are written in the `then` block of `we_are_jitted`. This hint function returns `True` after entering the trace. Therefore, the resulting trace has only call instructions to subroutines.

Figure 5.4 shows the how the method-traversal interpreter traverses a function body with respect to the bytecode, as denoted in Listing 4 and 5. In Figure 5.4, the gray-colored dotted line indicates a generated trace with the method-traversal interpreter. Normally, a tracing JIT only follows an executed side of the conditional branch. On the contrary, the tracer of threaded code generation follows both sides. To enable this, the method-traversal interpreter manages a hint called `traverse_stack`, which is represented as a stack data structure. It only stores program counters, so it is marked as *green* and finally removed from the resulting trace.

```
if opcode == JUMP_IF:
   target = ord(self.bytecode[pc])
   e = self.pop()
   if self._is_true(e):
     if we_are_jitted():
       pc += 1
       # save another direction
       traverse_stack = t_push(pc, traverse_stack)
     else:
       pc = target
   else:
     if we_are_jitted():
       # save another direction
       traverse_stack = t_push(target, traverse_stack)
     pc += 1
```

LISTING 6: Definition of JUMP_IF.

```
@dont_look_inside
def traverse_JUMP(self, pc):
    "A pseudo function for trace stitching"
    return pc

if opcode == JUMP:
  t = ord(self.bytecode[pc])
  if we_are_jitted():
    if t_is_empty(traverse_stack):
      pc = t
    else:
      pc, traverse_stack = traverse_stack.t_pop()
      # call pseudo function
      traverse_JUMP(pc)
  else:
    pc = t
```

LISTING 7: Definition of JUMP.

The behavior of the method-traversal interpreter can be explained with respect to the examples. The differences from a normal tracing JIT compiler are: (1) conditional branch, (2) back-edge instruction, (3) function call, and (4) function return.

Conditional branch

The tracer follows both sides of a conditional branch, first, tracing the `then` branch, and tracing `else` branch next.

When tracing a conditional branch ① in Figure 5.4, it saves the program counter that goes to another direction of a conditional branch to the `traverse_stack`. Listing 6 shows the handler for the `JUMP_IF`. Here, the traversal stack saves the another directions in lines 8 and 19.

```
@dont_look_inside
def traverse_RET(self,v):
  "A pseudo function for trace stitching"
  return v

if opcode == RET:
  if we_are_jitted():
    if t_is_empty(traverse_stack):
      return self.tla_RET(pc)
    else:
      pc, traverse_stack = traverse_stack.t_pop()
      traverse_RET(self.tla_RET(pc))
  else:
    return self.tla_RET(pc)
```

LISTING 8: Definition of RET.

Back-edge instruction

Upon a back-edge instruction, the tracer jumps to one of the remaining branches. It originally follows a back-edge instruction and finishes tracing when it reaches the beginning of tracing. Threaded code generation modifies such a behavior using the interpreter definition instrumented with hint instructions: it does not finish tracing until the tracer reaches the end of a target method and visits all its paths.

When tracing a back-edge instruction at ②, it does not follow the jump target. Instead, at ③, it pops a program counter from the traversal stack and goes to the other branch, which is an unfollowed branch of a previous conditional jump (E in Figure 5.3).

Seeing the implementation of `JUMP` in Listing 7, before jumping somewhere, it checks whether the traversal stack is empty or not. If empty, the tracer normally executes `JUMP`. Otherwise, it restores the saved program counter from the traversal stack and goes to that place. To tell the place of a back-edge instruction, we have to call the hint instructions `traverse_JUMP` and `traverse_RET`. The hint instructions are used in trace-stitching to restore the original control flow.

Function call

To reduce the compilation code size, threaded code generation does not inline a function call.

When tracing `CALL` instruction at ④, it does not follow the destination of `CALL` but emits a call instruction because subroutines are decorated with `dont_look_inside`.

```
[p0]
i1 = call_i(ConstClass(tla_DUP, p0))
i2 = call_i(ConstClass(tla_CONST_INT, p0, 1))
i3 = call_i(ConstClass(tla_GT, p0, 2))
i4 = call_i(ConstClass(_is_true, p0, 4))
guard_true(i4) [p0]
i5 = call_i(ConstClass(tla_CONST_INT, p0, 7))
i6 = call_i(ConstClass(tla_SUB, p0))
i7 = call_i(ConstClass(traverse_JUMP, p0))
i8 = call_i(ConstClass(tla_CALL, p0, 10))
i9 = call_i(ConstClass(tla_RET, p0, i8))
leave_portal_frame(0)
finish(i9)
```

LISTING 9: The trace generated temporarily from a method-transverse interpreter.

Function return

When tracing `RET` at ⑤, first, the tracer checks whether the `traverse_stack` is empty or not. If it is not empty, it restores a saved program counter and continues to trace. Otherwise, it executes `RET` instruction. The implementation is shown in Listing 8, and the behavior is almost the same as for `JUMP`.

Finally, the trace is obtained as shown in Listing 9. Note that it is still linear, so it will cut and stitch the generated trace to restore the original control flow.

5.2.4 Trace Stitching

The trace obtained by tracing the method-traversal interpreter is a linear execution path, because the tracer is led to track all paths by the interpreter. The trace-stitching technique enables for a correct execution by reconstructing the original control flow.

Figure 5.5 shows how trace stitching works and $\boxed{1}$ – $\boxed{5}$ indicate its working flow.

$\boxed{1}$**: the stitcher cuts where `cut_here` indicates** to handle each branch as a separate trace. In Figure 5.5, the stitcher cuts the node B that the hint instruction `traverse_JUMP` points to:

$\boxed{2}$**: the stitcher restores the conditional branch** by compiling the trace E – F as a bridge. When compiling as a bridge, the stitcher emits a label L and rewrites the definition of an original guard failure placed at B:

$\boxed{3}$**: the stitcher restores `JUMP` instruction** at the bottom of D. After that,

$\boxed{4}$**: it copies variables and instructions** that are not in the scope of the branch B – E – F for run-time correctness. Finally,

```
# Loop 1, token number is 13458300
[p0]
i1 = call_i(ConstClass(tla_DUP, p0))
i2 = call_i(ConstClass(tla_CONST_INT, p0, 1))
i3 = call_i(ConstClass(tla_GT, p0, 2))
i4 = call_i(ConstClass(_is_true, p0, 4))
guard_true(i4) [p0] # pointing to Bridge 1
i5 = call_i(ConstClass(tla_CONST_INT, p0, 7))
i6 = call_i(ConstClass(tla_SUB, p0))
# targeting to its own top
jump(p0, descr=TargetToken(13458300))

# Bridge 1, token number is 1345340
[p0]
i8 = call_i(ConstClass(tla_CALL, p0, 10))
i9 = call_i(ConstClass(tla_RET, p0, i8))
leave_portal_frame(0)
finish(i9)
```

Listing 10: Stitched traces. One linear trace is converted into one trace and one bridge, and are connected to a guard failure.

$\boxed{5}$**: the tailor folds or removes** constants or unused variables/instructions, respectively.

As a result, the trace tree is obtained, as shown on the right-most side of Figure 5.5. Within Multilevel RPython, the trace tree is represented as two traces, as shown in Listing 10. There is no linear trace, but one trace and a bridge are connected to a guard failure. If `guard_true(i4)` fails, the control goes to Bridge 1 and executes it.

Guard Patching in Trace Stitching

To successfully patch guard failures, the stitcher needs information on which guard failure goes to which bridges. Because the method-traversal interpreter depth first traverses an entire method, the relation between the guard failure and bridge can be resolved using a first-in-last-out method. Intending to resolve the relations between guards and bridges, *guard failure stack* is useful for managing each guard failure. Because the method-traversal interpreter manages the relation of branches by the stack data structure, the relations between the original loop and bridges are first-in-last-out. The guard failure stack saves guard failures in each guard operation and pops them at the start of a bridge, that is, right after the hint instructions `emit_JUMP` and `emit_RET` that indicate a cut-here line.

Figure 5.6 gives a high-level overview of how to resolve the connections between guard failures and bridges. When the resolver begins to stitch the trace, as shown in Figure 5.6, we sequentially read the operations from the trace. In node B, it picks up the guard `guard failure (g1)` and pushes it to the guard failure stack. In node C, it does the same thing in the case of node B. Next, in node E, it finishes reading an operation, before then starting to cut the current trace. During cutting the trace, it pops a guard failure and connects it to the bridge. In node F, it does the same thing as in the case of node F. In node D, it finally finishes reading and produces one trace and two bridges. The connections are illustrated as red arrows in Figure 5.6.

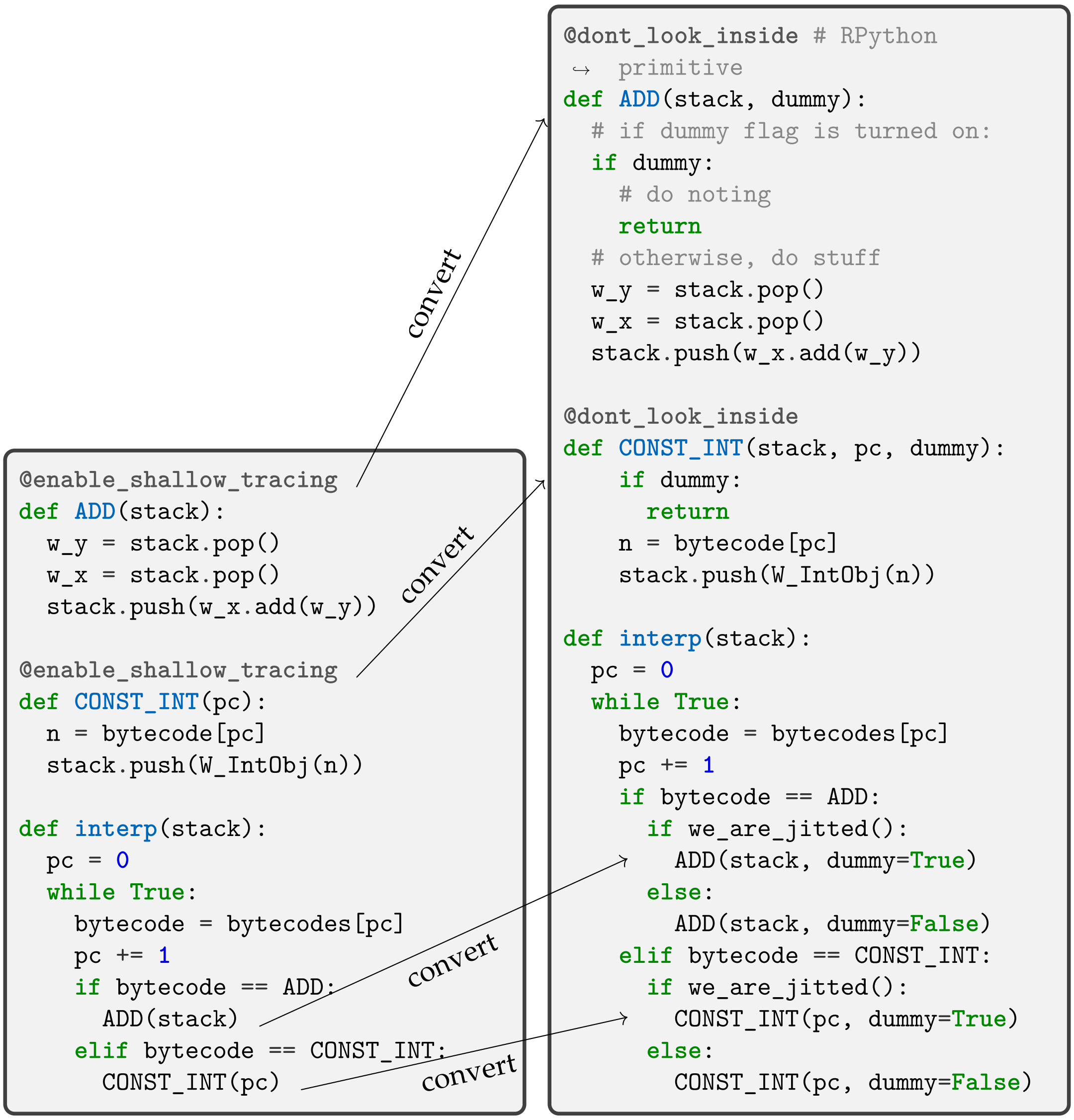
5.2.5 Shallow Tracing

When the threaded code generation technique is naively applied to a interpreter, the *inconsistency problem*, in which the tracer makes the interpreter's and global states inconsistent after tracing, will occur. This is caused by the nature of the tracer that executes the program during tracing. It is not a problem when the tracing compiler only follows the trace in actual program execution. However, with the threaded code generation technique, the behavior of the tracer is modified to pass through all paths, including those paths that are not executed, by a hint instruction in the method-traversal interpreter. This makes the interpreter's and global state (like the operand stack and heap) inconsistent, which will result in incorrectly compiled code. The problem is even worse when the interpreter performs global side effects, such as I/O operations and function calls.

The *shallow tracing technique* can avoid the problem. Shallow tracing allows the tracer to follow the code without also executing an interpreter. This technique is enabled by the hint decorator *enable_shallow_tracing*. This decorator lets the tracer not run it by transforming an interpreter definition. During interpreter transformation, the decorator adds a *dummy* flag to the last argument of each bytecode handler. The value of this flag is set to `True` during tracing, but after the trace has been recorded, it is rewritten to `False` in the trace.

This technique is explained by showing the example of an interpreter, as shown in the Listing 11. On the right-hand side, all handlers (except for `CALL`, `JUMP`, and `RET`) are defined as a method decorated with `enable_shallow_tracing`. On the right-hand side, all handlers and the interpreter are the results transformed by `enable_shallow_tracing`. Looking into the `ADD` handler, an extra flag `dummy` is inserted. Furthermore, in the fetch-decode-dispatch loop, the `dummy` is turned on when `we_are_jitted` is true. This `dummy` flag works as follows: during the tracing, it turned into `True` to do nothing but leave only a call instruction to the handler. Otherwise, `dummy` is `False` to run normally.

Next, the `dummy` flags should be turned off to make the resulting trace runnable. Listings 12 briefly explains the mechanism by which the code is changed. First, the Multilevel RPython tracer shallowly traverses all the paths of the function f, which is displayed at the top of Listing 12. Then, the trace is obtained as shown on the left side of the Listing 12 is obtained. Next, all additional flags `dummy` are deactivated to make the trace runnable. Since the positions of those flags are known, Multilevel RPython can automatically turn them off. Finally, the executable traces are obtained as shown on the right-hand side of Listing 12.



```
@enable_shallow_tracing
def ADD(stack):
  w_y = stack.pop()
  w_x = stack.pop()
  stack.push(w_x.add(w_y))

@enable_shallow_tracing
def CONST_INT(pc):
  n = bytecode[pc]
  stack.push(W_IntObj(n))

def interp(stack):
  pc = 0
  while True:
    bytecode = bytecodes[pc]
    pc += 1
    if bytecode == ADD:
      ADD(stack)
    elif bytecode == CONST_INT:
      CONST_INT(pc)
```

```
@dont_look_inside # RPython
↪ primitive
def ADD(stack, dummy):
  # if dummy flag is turned on:
  if dummy:
    # do noting
    return
  # otherwise, do stuff
  w_y = stack.pop()
  w_x = stack.pop()
  stack.push(w_x.add(w_y))

@dont_look_inside
def CONST_INT(stack, pc, dummy):
    if dummy:
      return
    n = bytecode[pc]
    stack.push(W_IntObj(n))

def interp(stack):
  pc = 0
  while True:
    bytecode = bytecodes[pc]
    pc += 1
    if bytecode == ADD:
      if we_are_jitted():
        ADD(stack, dummy=True)
      else:
        ADD(stack, dummy=False)
    elif bytecode == CONST_INT:
      if we_are_jitted():
        CONST_INT(pc, dummy=True)
      else:
        CONST_INT(pc, dummy=False)
```

LISTING 11: Overview of how the decorator `enable_shallow_tracing` works. The left-hand side is an interpreter that a language developer writes. The right-hand side is the actual executable interpreter after expanding `enable_shallow_tracing`.

5.3 Runtime Techniques for Multilevel Compilation

This section provides explanations for runtime techniques used in Multilevel RPython. First, the section provides the overview of how Multilevel RPython changes its compilation level at runtime. Second, the section details the taming of interpreters to realize the mechanism of changing a compilation level.

Multilevel RPython makes it possible to define the rule of a compilation level transition in an interpreter definition. Figure 5.7 provides and overview

```
def f(x):
    if x < 1:
        return x + 1
    return g(x)
```

```
# Just after shallow tracing.
# Flags are still activated.

# Loop 0
call_n(ConstClass('DUP'), p0, 1)
call_n(Const('CONST_I, 1'), p0, 1)
i0 = call_n(Const('LT'), p0, 1)
guard_true(i0) [p0]
call_n(ConstClass('CALL'), p0, 'g', 1)
p31 = call_n(Const('RET'), p0, 1)
finish(p31)

# Bridge 0
call_n(ConstClass('CONST_I, 1'), p0, 1)
call_n(ConstClass('ADD'), p0, 1)
p31 = call_n(ConstClass('RET'), p0, 1)
finish(p31)
```

```
# Just after deactivating flags.


# Loop 0
call_n(ConstClass('DUP'), p0, 0)
call_n(ConstClass('CONST_I, 1'), p0, 0)
i0 = call_n(ConstClass('LT'), p0, 0)
guard_true(i0) [p0]
call_n(ConstClass('CALL'), p0, 'g', 0)
p31 = call_n(ConstClass('RET'), p0, 0)
finish(p31)

# Bridge 0
call_n(ConstClass('CONST_I, 1'), p0, 0)
call_n(ConstClass('ADD'), p0, 0)
p31 = call_n(ConstClass('RET'), p0, 0)
finish(p31)
```

LISTING 12: Before and after applying shallow tracing to the function `f` are shown on the right-hand side of the Listing 11. During shallow tracing, all flags are activated (left side), but they are finally deactivated in the resulting trace (right side).

of the techniques to achieve multilevel compilation. The idea is to introduce a mechanism for the management of the lightweight and heavyweight compile interpreters. The mechanism is called *interpreter shifting*. The interpreter shifting defines the transition rule between each compilation level. The implementation of a transition is realized by the exception provided by RPython.

5.3.1 Implementaiton Details

Multilevel RPython uses a profiling-based approach to perform JIT compilation. The hint instructions for that purpose are `threaded_driver`, which is used for lightweight compilation, and `tracing_driver`, which is used for heavyweight compilation. The counter is incremented internally each time the `can_enter_jit` hint instruction provided by the respective driver is executed. When the threshold is exceeded, JIT compilation is initiated. The definition of both drivers are given in Listing 13.

Listing 14 shows how the transition of a compilation level can be implemented. In this example, the logic for the compilation level transition is implemented in the `JUMP_BACKWARD` instruction. To prepare, an array named `counts` is defined as a member in the `bytecode` class. This array is used to globally manage the counters in each `pc`. The threshold for transitioning to another level can also be defined by hand: in `interpret_threaded`, when the dispatch to `JUMP_BACKWARD` exceeds the defined threshold, `ContinueInTracingJIT` is thrown and the control moves on to `interpret_tracing` according to the programmed interpreter transition

```
threaded_driver = JitDriver(threaded_code_gen=True,
                            reds=["stack"], greens=["bytecode", "pc"])
tracing_driver  = JitDriver(reds=["stack"], greens=["bytecode", "pc"])

def interpet_threaded(..)
  tracing_driver.can_enter_jit(bytecode=bytecode,stack=stack,pc=pc)
  while True:
  opcode = bytecode[pc]; pc += 1
  if opcode == JUMP_BACKWARD:
    if bytecode.counts[pc] >= THRESHOLD:
      raise ContinueInTracingJIT(pc)
    bytecode.counts[pc] += 1
    ..
  elif ..

def interpet_tracing(..)
  while True:
  opcode = bytecode[pc++]
  if opcode == JUMP_BACKWARD:
    if bytecode.counts[pc] < THRESHOLD:
      raise ContinueInThreadedJIT(pc)
    pc = target
    tracing_driver.can_enter_jit(bytecode=bytecode,stack=stack,pc=pc)
   elif ..
```

LISTING 13: Overview of interpreters for lightweight and heavyweight compilation.

loop. Similarly, if `ContinueInThreadedJIT` is thrown in `interpret_tracing`, a transition to `interpret_threaded` occurs based on the written rules.

5.4 Optimization for Threaded Code Generation with Interpreter in the Meta-Tracing JIT Compiler

As well as adding a new compilation level to a meta-tracing JIT compiler, it is possible to implement optimization for the added compilation behavior by using an interpreter. To optimize threaded code generation, inline caching [Deutsch and Schiffman, 1984] is implemented. This section explains how to implement this technique on the method-traversal interpreter.

5.4.1 Inline Caching in Method-Traversal Interpreter

Inline caching [Deutsch and Schiffman, 1984] is an optimization technique that was first developed for Smalltalk. The goal of inline caching is to realize a fast method lookup at runtime by remembering the class of a callee, which is a receiver in the context of Smalltalk, at every call site.

```
# user-defined threshold to shiftup the compilation level
THRESHOLD = 1539


class ContinueInTracingJIT(Exception):
  def __init__(self, pc):
    self.pc = pc


class ContinueInThreadedJIT(Exception):
  def __init__(self, pc):
    self.pc = pc

def interpreter(..):
  pc = 0
  while True:
    try:
      w_r = interpret_threaded(pc, ..)
      return w_r
    except ContinueInTracingJIT as e:
      pc = e.pc

    try:
      w_r = interpret_tracing(pc, ..)
      return w_r
    except ContinueInThreadedJIT as e:
      pc = e.pc
```

Listing 14: Overview of the interpreter shifting mechanism. The left-hand and right-hand sides show the definitions of JIT policy-shifting exceptions and the interpreter shifting loop, respectively.

In general, calling a dynamically bound method takes longer time than calling a statically-bound method because a system needs to look up a correct method by using the runtime class of a callee method. A same kind of problem occurs in threaded code generation. For example, there is the case in which a method invocation to the compiled method `f` belonging to the class `A` is going to be performed. As shown in Figure 5.8, at the point of calling a method from a compiled trace, the control is transferred from the trace to the interpreter through the `handler_CALL` method. In the interpreter, `jit_merge_point` looks up the correct callee trace and let the control transfer to it. Because this method invocation needs at least these steps, it becomes overhead compared with directly calling a compiled method.

An inline caching technique helps the system look up a correct callee method faster, even in the case of threaded code generation. Inline caching stores the looked-up callee method address at the call site, for example by overwriting the call instruction. The type of a looked-up callee method could be changed: it should check the runtime just before directly jumping to the compiled code (and if the test fails, the control should call the looked-up method) [Hölzle, Chambers, and Ungar, 1991].

```
def interp(method, stack, ..):
  while True:
    jit_merge_point(..)
    opcode = bytecode[pc++]
    if opcode == CALL:
      handler_CALL(stack, ..)

def handler_CALL(stack, ..):
  method = pop(stack)
  r = interp(method, stack, ..)
  push(r, stack)
```

LISTING 15: Interpreter definition of a dispatch loop and a handler for CALL.

```
...
call(hanlder_CALL, ...)
...
```

LISTING 16: Produced trace at tracing CALL instruction with the interpreter shown in Listing 15.

In threaded code generation, it is possible to realizing the inline caching technique by using hint instructions inserted into an interpreter. If `handler_CALL` is naively implemented, it will be written as in Listing 15. The generated trace, which is shown in Listing 16, always calls the interpreter and enters the method lookup routine to perform method invocation. To improve this routine through the use of inline caching, the hint instructions called `record_type`, `check_type`, and `call_assembler` are used. The overview and example of an interpreter definition and resulting traces are shown in Figure 5.9 and Listings 17 and 18. At interpreting, `record_type` records the runtime type of a looked-up callee method. At tracing, when the meta-tracing JIT compiler traces the virtual instruction, the compiler emits two traces: fast and slow paths. The fast path trace directly calls a compiled method, but the slow path calls `handler_CALL`. In the slow path, the control goes back to the merge point placed just after the fast path. `check_type` validates that the runtime type of a callee method equals the stored type by `record_type`. If the validation fails, the control goes to the slow path. `check_type` is compiled into `guard_ptr_eq`, as shown in the left-hand side of Listing 18. `call_assembler` is compiled into the direct call instruction in the fast path, as shown on the left-hand side of Listing 18.

5.5 Preliminary Evaluation Using Simulated Threaded Code Generation

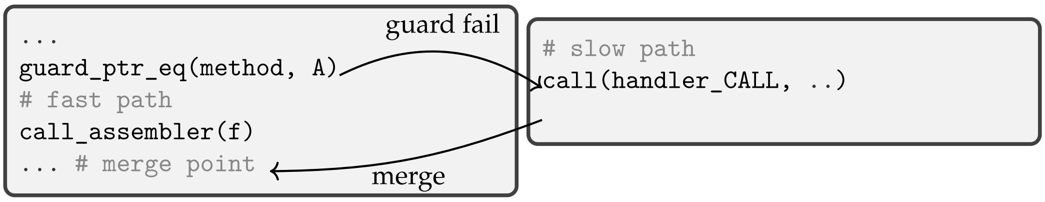
This section experimentally evaluates the potential performance of our threaded code generation by simulating the behavior with PyPy. Here, it

```
def interp(method, stack, ..):
  while True:
    jit_merge_point(..)
    opcode = bytecode[pc++]
    if opcode == CALL:
      # check whether the runtime type of this method
      # equals to the recorded runtime type
      if check_type(method, pc):
        method = pop(stack)
        # if it matches, calling the method with call_assembler
        call_assembler(interp(method, stack, ..))
      else:
        # otherwise, call handler_CALL
        handler_CALL(stack, ..)

def handler_CALL(stack, ..):
  method = pop(stack)
  if not we_are_jitted():
     # during interpreting (not jitted),
     # record runtime method types at pc
     record_type(pc, method.type)

  r = interp(method, stack, ..)
  push(r, stack)
```

LISTING 17: Interpreter definition, which is based on Listing 15, instrumented to enable inline caching.



```
...
guard_ptr_eq(method, A)
# fast path
call_assembler(f)
... # merge point
```

```
# slow path
call(handler_CALL, ..)
```

LISTING 18: Produced traces at tracing CALL instruction with virtual instructions to enable inline caching.

compares the performance of threaded code generation with the interpreter.

Section 5.2 describes threaded code generation that enables a lightweight compilation with a meta-tracing JIT compiler. The question arises of whether the technique is effective at runtime or not. To answer the question, this section measured JIT compilation time and code size of traces of PyPy's tracing JIT compiler and our simulated threaded code generation (SMTG), respectively. In addition, the section compared the potential performance of the two following executions: PyPy 3.7 with SMTG and interpreter-only execution.

5.5.1 Simulated Threaded Code Generation (STCG) in PyPy

To measure the potential performance of our threaded code generation, it is needed to reproduce its behavior on PyPy. The brief ideas of the *simulated* *threaded code generation (STCG)* are;

Idea 1. All subroutines are not inlined, but call instructions to subroutines are left.

Idea 2. Tracing all paths of a target program area *at once*.

Idea 1 can be easily reproduced by manually adding `dont_look_inside` to the PyPy interpreter manually. The problem is how to reproduce idea 2. The current PyPy does not have such a function, but it has a guard failure. When the number of failing a guard surpasses a threshold, a tracing JIT starts to trace the destination of a guard and connects the original trace and the generated trace from a guard. Then, if it runs programs with enough time, all runtime paths are eventually traced by a guard failure. Therefore, it is possible to reproduce the behavior of idea 2 by running the benchmarks for a long time.

5.5.2 Setup

This section explains the environment and how the preliminary experiments were performed.

System

The evaluation and experiment are conducted the preliminary benchmark on the following environment; CPU: Ryzen 9 5950X, Mem: 32GB DDR4-3200MHz, OS: Ubuntu 20.04.3 LTS with a 64-bit Linux kernel 5.11.0-34-generic.

Implementation

The original PyPy 3.7 versioned 7.3.5[5] is used, and the STCG[6] is also implemented with the PyPy implementation.

Programs for Experiments

You can find all benchmark programs here[7]. All benchmarks that can be executed without any other libraries are chosen. Especially, `fib` and `tak` are programs causing the path-divergence problem.

Methodology

Two experiments are conducted on PyPy's original micro-benchmark suite plus our original ones;

---

[5] `https://downloads.python.org/pypy/pypy3.7-v7.3.5-linux64.tar.bz2`

[6] `https://foss.heptapod.net/pypy/pypy/-/tree/branch/py3.7-hack-measure-bytecode-dispatch`

[7] `https://foss.heptapod.net/pypy/benchmarks/-/tree/topic/python3_benchmarks/bitbucket-pr-5`

Experiment 1. Measuring the overhead of tracing and compilation in our STCG.

Experiment 2. Measuring the stable speeds of our STCG.

Experiment 1. To measure the overhead of tracing and compilation, PyPy 3.7–7.3.5 with a tracing JIT and our STCG is used. Their compilation time and the size of the traces to compile are measured, and they are normalized to PyPy with a tracing JIT. The compilation time includes the tracing. Note that the implementation of our full-fledged threaded code generation on PyPy is ongoing, so the behavior is simulated (Section 5.5.1 describes how to do it). The results are shown in Figure 5.10.

Experiment 2. Comparison of stable speeds between the STCG on PyPy 3.7 with the interpreter-only execution. Interpreter-only execution means that JIT compilation is turned off by passing `--jit off` when running scripts. The averages and standard deviations of the STCG normalized to interpreter-only execution are calculated. The results are shown in Figure 5.11.

The maximum iteration count is 100 from the results that plot the related speed-up ratio in the STCG and PyPy's tracing JIT as shown in Figure 5.12. From these results in the STCG, it is confirmed that almost all programs except those reach their stable state after the fifth iteration. In addition, PyPy's tracing JIT reaches its stable speed after the 30th iteration. Experiment 1 requires a number of operations and times for compilation, and experiment 2 needs STCG's stable speed; in this context, we decide that the max iteration count 100 is enough to reach the stable speed. Thus, in experiment 2, the first five iterations are excluded for calculating the average value of the stable speed of every program.

5.5.3 Results of Experiment 1: The Overhead of Our STCG

The objective of this experiment is to potentially evaluate the start-up time of simulated threaded code generation. The results are shown in Figure 5.10. On average, in the case of trace sizes to compile, PyPy 3.7 with STCG is about 78% smaller than PyPy 3.7–7.3.5 with a tracing JIT, and 13 of 17 programs are about 50% smaller than PyPy's tracing JIT. Furthermore, in the case of compilation time, PyPy 3.7 with STCG is about 60 % shorter than PyPy 3.7–7.3.5 with a tracing JIT. Here, 13 of 17 programs, which are the same as in the case of the size of traces to compile, are 60 % shorter than PyPy's tracing JIT. However, PyPy 3.7 with STCG size of traces and the compilation time on `nbody` is almost the same as that of PyPy's tracing JIT. This program computes the N-body simulation with a matrix calculation. This calculation is implemented as a big for loop, so there is less of an effect on performing threaded code generation than full-optimized tracing.

5.5.4 Results of Experiment 2: The Stable Speed

The results are summarized in Figure 5.11 (their values in every iteration are shown in Figure 5.12). The results of the PyPy 3.7 simulated threaded code generation are normalized to interpreter-only execution. On average, STCG is 7% faster than the interpreter only. PyPy 3.7 with STCG is over 4% faster in 9 of the 17 benchmarks, and ± 3% faster in 8 of the 17 benchmarks. In particular, `meteorcontest` and `nqueens` are about 27% to 34% faster than the interpreter.

5.5.5 Discussion

In experiment 1, there is a relation between the size of the traces and compilation time. Simulated threaded code generation can reduce the size of the traces and compilation time, so we can use it to reduce the warm-up time.

Furthermore, programs with the path-divergence problem (`fib` and `tak`) are at least 96% smaller and 80% faster in trace size and compilation time, respectively. In general, when the path-divergence problem occurs, retracing often occurs, and too many traces overlap each other and lead to high overhead in run-time performance. However, the result shows that the STCG traces and compiles only a primary hot function, so the trace sizes and compilation time are much smaller and shorter than a tracing JIT. Thus, a method-based threaded code can reduce the trace size and compilation time.

From both experiments, it can be inferred that our method-based threaded code generation will bring some benefits to startup performance. To make the technique more effective, it is necessary to select functions that have a structure similar to `meteorcontes` and `nqueens` as well as programs with the path-divergence problem. In those programs, much part of one primary solver function with complex conditional branches is executed inside the main loop, but the other functions are not. In other words, during solving conditions, instead of running a single main region over and over, some regions are sometimes run randomly. This execution model potentially causes the path-divergence problem. Thus, method-based threaded code generation can work effectively on such programs. To enhance the effectiveness of our method-based threaded code generation with this assumption, we need to select programs with complex conditional branches inside a long iteration in addition to programs that indeed cause the path-divergence problem.

Limitation of Threaded Code Generation. Threaded code generation is placed at the initial compilation level, so the compilation limits further optimizations. For example, because the compilation does not inline an instruction handler but leaves the call instruction. There are gaps between this lightweight JIT compilation and the tracing JIT that RPython provides. Thus, this gap suggests that several optimizations are needed between the lightweight and the tracing JITs.

To allow for further optimizations such as allocation removal, implementing higher levels of lightweight JIT compilation is a good plan. For example, the level-2 lightweight JIT just inlines a stack manipulation, but other operations do not. The level-3 lightweight JIT inlines auxiliary methods, but others are not. We are able to realize these levels at low cost by placing `dont_look_inside` into each method header.

5.6 Evaluation and Experiments in PySOM and Multilevel RPython

The question of whether the mixture of different JIT compilation levels is effective in improving the performance of real-world applications remains to be solved. To address the question, this section evaluates and examines the effectiveness of multilevel JIT compilation in large-scale benchmark programs in initial performance. Evaluation and experiment take two-level compilation, where threaded code generation is used as lightweight JIT compilation and the (meta-)tracing JIT compiler is responsible for heavyweight JIT compilation.

This section first evaluates the performance of threaded code generation by using tiny and microbenchmark programs. The performance characteristics of threaded code generation are discussed in its compilation time and code speed on a steady state, compared to tracing JIT and interpreter execution in Section 5.6.1.

Next, this section has an experiment for the two-level multilevel JIT compilation. Unlike the micro-benchmark evaluation, it uses large-scale programs to address realistic workloads. Because PySOM does not support to read image files generated by other Smalltalk implementations and the entire Smalltalk instruction set, it is difficult to run a real-world application on PySOM. Given this context, the experiment simulates an application that has a real-world workload by combining larger benchmark programs such as the classic constant solver, simulation of an operating kernel system, or Json parsing. Section 5.6.2 discusses the performance and capability for large-scale applications of multilevel JIT compilation.

Threat to validity The tested language PySOM is originally well implemented and capable of running non-trivial applications, but the results may change if tested on production-level languages like PyPy.

5.6.1 Microbenchmark Evaluation

The objective of this evaluation is to discuss if threaded code generation, which is useful as lightweight JIT compilation alongside heavyweight tracing JIT compilation. This section presents the results and discussion obtained from this experiment.

The microbenchmark evaluates the compilation time and peak performance in a steady state of threaded code generation and tracing JIT compilation in Multilevel RPython. To begin with, by comparing the compilation time of threaded code generation with tracing JIT, it is possible not only to understand the characteristics of compilation, but also to discuss how much runtime overhead caused by dynamic compilation can be reduced using lightweight compilation. In addition, a peak performance measurement is needed to measure the benefit of compiling with threaded code generation. By combining these two metrics, it is possible to discuss the benefit and cost of performing threaded code generation as lightweight JIT compilation.

Setup

PySOM[8] is used as the target language. PySOM is an implementation of a Smalltalk dialect called Simple Object Machine (SOM) [Haupt et al., 2010] by RPython. PySOM is a language implementation consisting of about 14,000 lines of Python source code, and it is suited for demonstrating the effectiveness of our proposal with a realistic benchmark set. As shown in one study [Marr, Daloze, and Mössenböck, 2016], SOM can run as complex programs as other languages such as Ruby or Java.

The generation of threaded code and multilevel JIT compilation are implemented in the PySOM interpreter. The number of lines of source code required for the implementation is about 2400 lines.

The experiments are run on a single-socket 16-physical core Ryzen 9 5950X CPU at 3.4 GHz, running the Ubuntu 22.04 operating system. The benchmark machine has 32 GB RAM, which is enough to prevent memory swapping. Experiments are performed with ReBench [Marr, 2018], which is a tool to run and report benchmark experiments. Using ReBench allows not only to reproduce experiments and documenting all benchmark parameters, but also to minimize noise caused by system interference.

Methodology

To evaluate threaded code generation, the peak performance at steady state is measured using microbenchmark programs. To perform this evaluation, each microbenchmark program was run for 2000 times as one set. Here, 2000 iterations can be considered a long enough number that the compilation time is negligible in determining the steady-state velocity. This set was iterated 30 times. After calculating the geometric mean for each iteration of the 30 sets, the overall geometric mean was calculated at the end. Furthermore, to measure the compilation time of threaded code generation and tracing JIT compilation, the PyPy logging system is used that can report how much time is consumed to tracing and compiling traces.

The benchmark programs were collected from Are We Fast Yet? benchmark and SOM's original benchmark programs. They can be classified into tiny,

---

[8] `https://github.com/SOM-st/PySOM`

Type	Benchmark	Description
Macro	CD	Simulation of an airplane collision detector. Based on WebKit's JavaScript CDjs
	Richards	Classic benchmark simulating an operating system kernel
	DeltaBlue	Classic constraint solver used to tune, e.g., Smalltalk, Java and JavaScript VMs.
	Json	JSON string parsing benchmark
Micro	Fannkuch	Program defined by programs in [Anderson and Rettig, 1994]
	Bounce	Simulating a ball bouncing within a box
	Permute	Generating permutations of an array
	Queens	Solving the eight queens problem
	List	Recursively creating and traversing lists.
	Storage	Creates and verifies a tree of arrays to stress the GC
	Sieve	Finding prime numbers based on the sieve of Eratosthenes
	BubbleSort	Sorting algorithm that repeatedly swaps adjacent elements
	QuickSort	Sorting algorithm classified into divide-and-conquer algorithm
	TreeSort	Sorting algorithm that builds a sorted binary tree and traverses it
	Mandelbrot	Calculating Mandelbrot set
Tiny	Fibonacci	Calculating a fibonacci number
	Dispatch	Repeatedly invoking one tiny method
	Loop	Repeatedly counting up a number
	Recurse	Repeatedly invoking a method recursively
	Sum	Calculating the sum from 1 to a given number

TABLE 5.1: Benchmark descriptions.

micro-, and macro-benchmarks according to the size of the application. In this micro-benchmark evaluation, tiny and microbenchmarks are used. The descriptions of these benchmark programs are shown in Table 5.1.

Code Sizes and Compilation Times

Figure 5.13 shows the graph plotting the bytecode sizes and the compilation times consumed. Table 5.2 shows the bytecode sizes of micro-benchmark programs. The results show that threaded code generation increases the compilation time linearly based on bytecode size. Meanwhile, tracing JIT has spikes in `List` and `QuickSort`. Regression lines are calculated as follows:

$$y_{\text{threaded code}} = 0.0044 \times x + 0.31 \;\; (R^2 = 0.7087) \tag{5.1}$$

$$y_{\text{tracing JIT}} = 0.411x + 5.0 \;\; (R^2 = 0.118) \tag{5.2}$$

Given each R^2 in their regression lines (0.7087 in threaded code, 0.118 in tracing JIT), there is a correlation between compilation time and code size for threaded code generation, but tracing JIT shows unstable characteristics between the compilation time and code size. Furthermore, the slope of the regression line for threaded code generation (4.42×10^{-3}) is lower than that for JIT trace (0.411).

These results indicate that using threaded code generation for warm spots, which are executed less frequently than hot spots, is likely to have negligible compile overhead. However, small functions or code fragments that are

Micro Benchmark	**Size (byte)**
Dispatch	56
Fibonacci	59
Sum	86
Recurse	111
Loop	112
Sieve	138
List	246
Storage	272
Queens	285
Mandelbrot	422
Fannkuch	474
Bounce	488
BubbleSort	600
TreeSort	702
QuickSort	749

TABLE 5.2: Micro benchmark programs and their bytecode sizes. Standard libraries of the PySOM system is excluded.

executed only a few times with an execution speed of less than 0.3 ms incur compile overhead and should be executed by an interpreter. That being said, tracing JIT is difficult to predict the compile cost, and even if the code size is small, as in List and QuickSort, the compile cost is very large compared with threaded code generation. Therefore, it is necessary for multilevel compilation to be compiled only for hot spots (or hot loops) that are frequently executed.

Peak Performance at Steady State

Figure 5.14 shows the peak performance in steady state. In general, threaded code generation is 5% faster than interpreter execution, but 94% slower than tracing JIT execution. The reason why threaded code is much slower than the tracing JIT is that it does not perform many optimizations for the obtained traces, because the trace consists of only essentially call instructions to handlers. It is trade-off because the cost of the compile time is so small compared to tracing JIT.

Next, the characteristics of threaded code generation are discussed based on the nature of each benchmark. In particular, recursive call-intensive programs such as `Fibonacci`, `Recurse`, and `Sum` are from 10% to 24% faster than the interpreter execution. This is because recursive call instructions are converted into direct calls due to inline caching optimization. In addition, for call intensive programs, `TreeSort` performs 5% better in threaded code generation than interpreter execution, but other programs like `Bounce`, `Queens`, `QuickSort` and `Storage` are about 10% slower than interpreter execution. This is because inline caching works in `TreeSort` but not effectively in the others. Here, `Bounce`, `Queens`, `QuickSort` and `Storage` have failed to apply inline caching to all function calls. As a result, some function calls have become slow function calls. Furthermore, loop-intensive programs like `BubblerSort`, `Fannkuch`, `Loop`, `Mandelbrot`, and `Sieve` are from 5% to 17% better than interpreter execution.

According to these results, to use threaded code generation as a lightweight JIT compilation, it should be applied to call intensive, especially recursive call intensive, programs. Threaded code generation would also benefit from being applied to warm, but not hot, loops.

5.6.2 Multilevel JIT Experiment

Generally, warm-up time is considered a bottleneck in many real-world applications. For example, when considering a client application, the application is often initialized first and then, the main routines are executed. In this case, if heavyweight JIT compilation is performed at the initialization stage, it is a waste to perform heavyweight JIT compilation because the initialization part is not used in subsequent executions. Instead, it is more beneficial to do lightweight compilation such as threaded code generation. This section validates whether multilevel JIT compilation, which combines threaded code generation and tracing JIT, works effectively for large benchmark programs and applications with realistic workloads.

Setup

The same machine and PySOM implementation with Multilevel RPython is used, as shown in Section 5.6.1, but the difference is that the multilevel JIT compilation feature is enabled on it.

Methodology

The multilevel JIT experiment evaluates the startup performance and peak performance in multilevel JIT compilation using threaded code generation as lightweight compilation and tracing JIT compilation as heavyweight compilation. In multilevel compilation, threaded code generation is performed first, and tracing JIT is performed later.

To simulate a real-world workload using benchmark programs, macro-benchmark programs originally prepared in PySOM were combined. `Richards`, `Json`, `Deltablue`, and `CD` were used for this simulated real-world workload application. Figure 5.15 shows the summarized number of method invocations in each macro benchmark program. The simulated application has a long tail distribution of the number of method invocations. This experiment was conducted on the application. The experiment was iterated once in one set, and the set was iterated 2,000 times.

Given the result in Section 5.6.1, the author sets up a model where heavyweight tracing JIT could be applied to hot loops and lightweight threaded

Strategy	Threshold for Loop	Threshold for Function Call
Multilevel	1539	2003
Tracing Only	1039	1697
Threaded Only	1539	2003

TABLE 5.3: The threshold for the multilevel JIT experiment.

code generation is applied to method invocation intensives as a compilation strategy in multilevel JIT compilation. The numbers used in the experiment are shown in Table 5.3, and the visualization of which methods are compiled at which level according to the threshold set. in Figure 5.16. These numbers are determined based on the heuristics that when looking at the distribution of method calls, heavyweight tracing JIT compiles about 20% of the total, and lightweight threaded code generation compiles 30%. The percentage was determined by reference to the results of the paper about DaCapo benchmark [Blackburn et al., 2006], where approximately 30% of all defined functions were compiled in lightweight and 10 % in heavyweight. Visualization is shown in Figure 5.16.

In Table 5.3, the values for tracing only is the default numbers already set by the PyPy developers. In addition to multilevel JIT execution, tracing JIT-only, tracing JIT-only (the thresholds are the same as for multilevel JIT execution), threaded code generation-only, and interpreter-only executions can be used for comparing performance.

Results of the Experiment and Discussion

Figure 5.17 shows the results of this experiment on the simulated real-world workload application. Overall, multilevel JIT compilation is about 14% faster than tracing JIT only execution and about 7% faster than tracing JIT execution, the threshold of which is the same as multilevel JIT execution.

From these results, to speed up an application where the distribution of method calls is long-tailed, it is not enough to raise or lower the tracing JIT threshold: rather, new lightweight JIT compilation such as threaded code generation needs to be added. However, threaded code execution is about 3% slower than interpreter-only execution. This means that the optimization currently implemented, which is inline caching, for threaded code generation is not enough to make lightweight JIT compilation more effective. Implementing techniques such as superinstructions [Ertl et al., 2002] for threaded code generation could be implemented, but this is left for future work.

5.7 Related work

There are many techniques that can be used to improve the warm-up performance in a VM. Similar approaches to our work include improving the runtime performance of an interpreter and implementing a lightweight JIT compiler in a VM. In contrast, another approaches have tried to realize an ahead-of-time (AOT) compilation for managed language runtimes such as Java to improve its startup performance. Also, regarding the attempt to add new behaviors to the meta-JIT compiler , the work on Eclipse OMR and RPython is close to our approach.

5.7.1 Improving an Interpreter's Performance

The continuation passing style (CPS) [Steele, 1977] is a technique to improve an interpreter's performance. Whereas threaded code generation removes conditional jumps, CPS can perform tail-call optimization that converts function calls into jumps because all calls are tail-calls in CPS, which has the advantage of not increasing the stack in an interpreter.

The superinstruction [Casey et al., 2003] allows us to reduce the number of indirect jumps by extending the VM instruction set with a superinstruction that corresponds to the work done by multiple instructions in a single instruction. Threaded code generation can be applied to an interpreter written in RPython without modifying VM instruction sets, but superinstruction has a limitation because the VM instruction encoding may not allow for the modification, and the optimal instruction sets can be combined as superinstruction may be changed depending on the workload.

5.7.2 Template JIT Compilation

Using a lightweight JIT compiler is effective when improving the warm=up performance in a VM. A lightweight JIT compiler only applies optimizations that do not take a significant amount of time to compile and do not increase the binary size too much.

Although the threaded code generations produces threaded code based on traces at runtime, template JIT compilers [Iliasov, 2003; Wimmer et al., 2013] use a heavyweight compiler to emit machine code by aggregating machine code parts that are generated at build-time. They always produce the same machine code for a given input. The threaded code technique is also used in template JIT compilers [Ertl and Gregg, 2003; Piumarta and Riccardi, 1998].

In recent years, several template-based JIT compilers [Coffin et al., 2020; Xu and Kjolstad, 2021] have been proposed. Micro JIT [Coffin et al., 2020] is a template JIT compiler for resource constrained environments like the Internet of Things (IoT). Micro JIT is built on Ecplipse OpenJ9 [Eclipse Foundation, 2017], alongside its JIT compilers. Copy-and-patch compilation [Xu and

Kjolstad, 2021] is also based on a template-based approach, and its uniqueness is its fast compilation time compared with LLVM `-O0` and the Lifoff, which is a lightweight WebAssembly JIT compiler made by Google.

5.7.3 Ahead-of-Time Compilation

Another approach to achieve fast startup performance is to use ahead-of-time (AOT) compilation for managed language runtimes [Krylov et al., 2021; Wimmer et al., 2019]. GraalVM Native Image [Wimmer et al., 2019], which is a part of the GraalVM project, is capable of compiling Java programs into machine code at build time. It can achieve a performance improvement of up to two orders of magnitude compared with Java HotSpot VM while maintaining good peak performance. Eclipse OMR also has an AOT compiler for WebAsembly [Krylov et al., 2021], hence showing competitive performance compared with the performance of the V8 engine.

5.7.4 Introducing a New Behavior into a Meta-JIT Compiler

Some studies have allowed to realize a new optimization behavior in a meta-JIT compiler. In contrast to our approach that introduces a new compilation/optimization behavior, [Nanjekye, Bremner, and Micic, 2021] proposed an Eclipse OMR-based approach that introduces a new GC for RPython. Eclipse OMR [Eclipse Foundation, 2019] is a reliable component that includes a heavyweight JIT compiler and a dedicated GC for building language runtimes. To prove its reusability, Ruby+OMR, SOM++ Smalltalk and an experimental version of CPython have been built by using Eclipse OMR components.

5.8 Conclusion

This chapter has introduced Multilevel RPython, which can generate a single compiler that can behave as a method-based lightweight and trace-based heavyweight JIT compilers, and both. The original RPython performs (meta-)tracing compilation, but its other behavior as threaded code generation is achieved by controlling the behavior of a meta-tracing compiler by the hint instructions inserted in an interpreter.

In the preliminary evaluation for threaded code generation, the experiments conducted in PyPy has shown that threaded code can reduce the size of traces and the compilation time by approximately 80% and 60%, respectively. It can run 7% faster than the interpreter-only execution in when there is stable speed.

In the evaluation of the effectiveness of multilevel JIT compilation in warm-up performance, the experiments show that using threaded code generation as an initial compiler improves the overall performance about 14% in the application simulating a real-world workload.

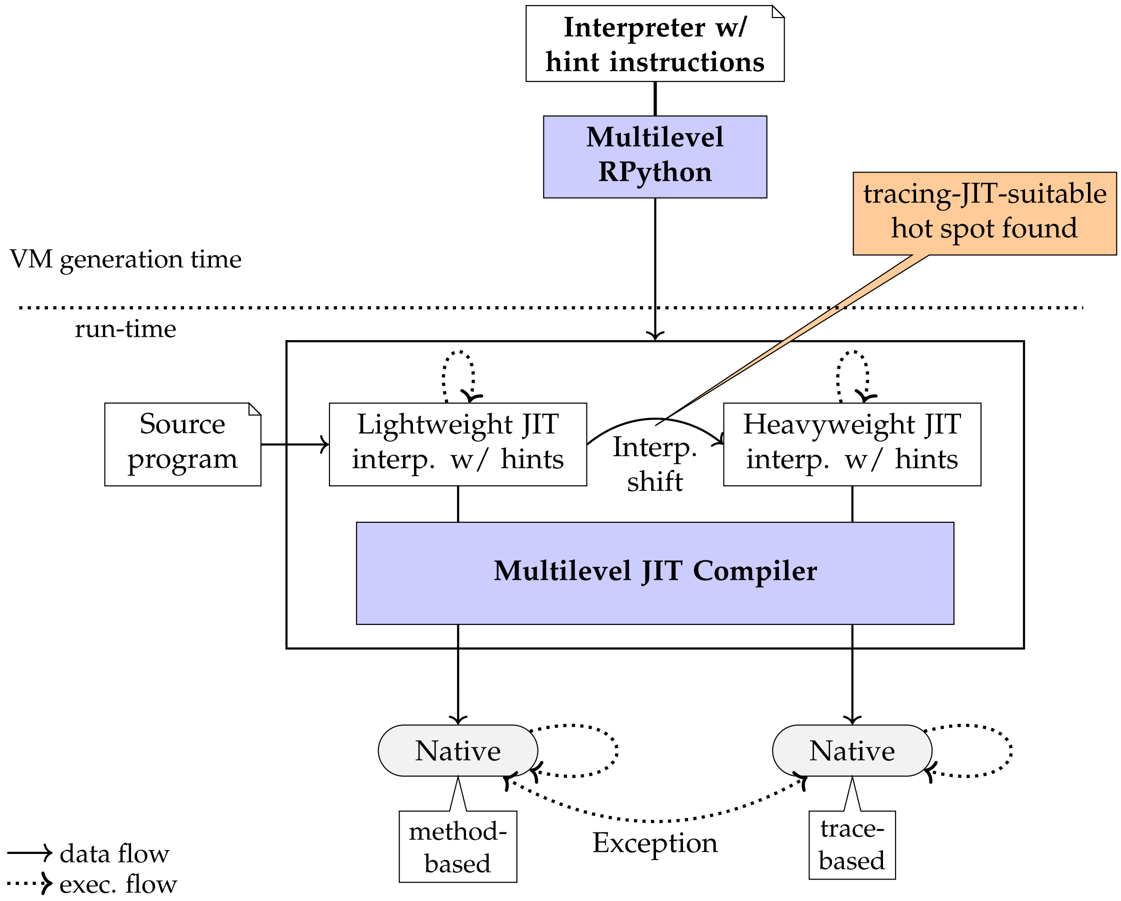
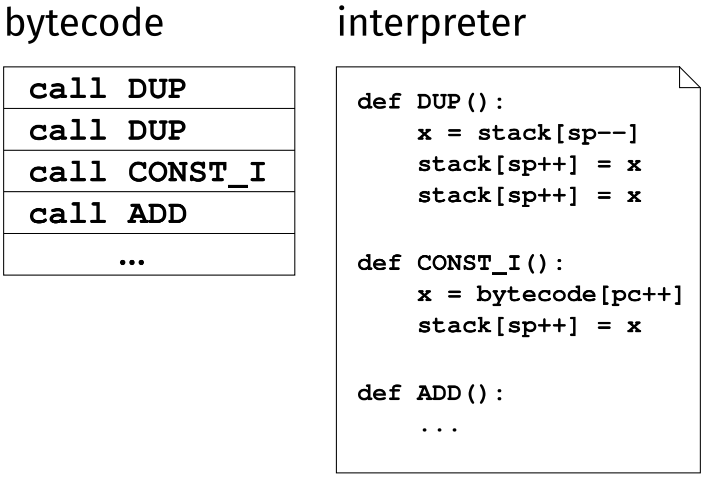

Interpreter w/ hint instructions
Multilevel RPython
tracing-JIT-suitable hot spot found
VM generation time
run-time
Source program
Lightweight JIT interp. w/ hints
Interp. shift
Heavyweight JIT interp. w/ hints
Multilevel JIT Compiler
Native
Native
method-based
Exception
trace-based
data flow
exec. flow

FIGURE 5.1: Overview of Multilevel RPython: there exists two different interpreters in the generated VM. At VM generation time, Multilevel RPython generates a VM from an interpreter instrumented with hint instructions used for multilevel compilation. At runtime, the different two interpreters are used for lightweight and heavyweight compilations. First, a source program is run on the interpreter for lightweight compilation. When hot spots are detected, the execution is switched to the interpreter for heavyweight compilation from one for lightweight compilation. This transition is performed by the interpreter shifting technique described in Section 5.3.


bytecode
interpreter

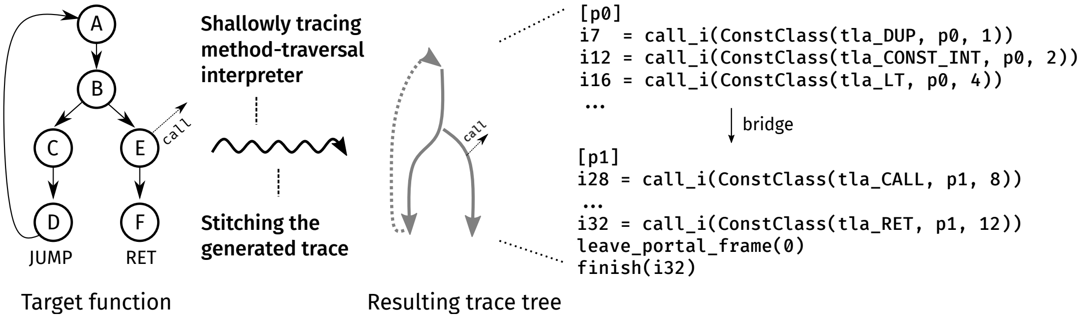
```
call DUP
call DUP
call CONST_I
call ADD
...
```

```
def DUP():
    x = stack[sp--]
    stack[sp++] = x
    stack[sp++] = x

def CONST_I():
    x = bytecode[pc++]
    stack[sp++] = x

def ADD():
    ...
```

FIGURE 5.2: An overview of how threaded code works.


A
B
C
E
D
F
call
JUMP
RET
Target function
Shallowly tracing method-traversal interpreter
Stitching the generated trace
call
Resulting trace tree
bridge

```
[p0]
i7  = call_i(ConstClass(tla_DUP, p0, 1))
i12 = call_i(ConstClass(tla_CONST_INT, p0, 2))
i16 = call_i(ConstClass(tla_LT, p0, 4))
...
[p1]
i28 = call_i(ConstClass(tla_CALL, p1, 8))
...
i32 = call_i(ConstClass(tla_RET, p1, 12))
leave_portal_frame(0)
finish(i32)
```

FIGURE 5.3: A sketch of how RPython method-based lightweight JIT compiler works. From the target function in the left-hand side, it generates the trace tree shown in the right-hand side.

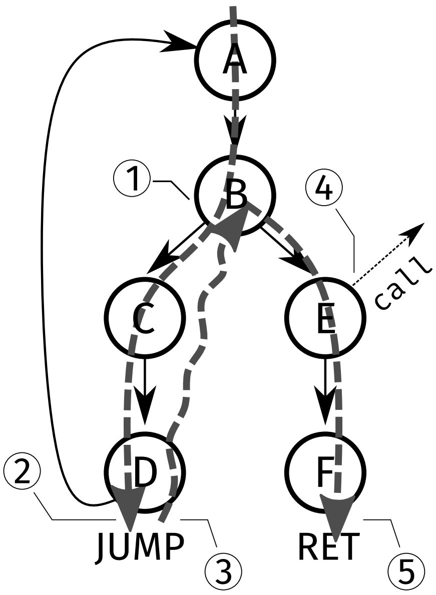

A
①
B
④
C
E
call
②
D
F
JUMP
③
RET
⑤

FIGURE 5.4: Tracing the entire of a function with method-traversal interpreter.

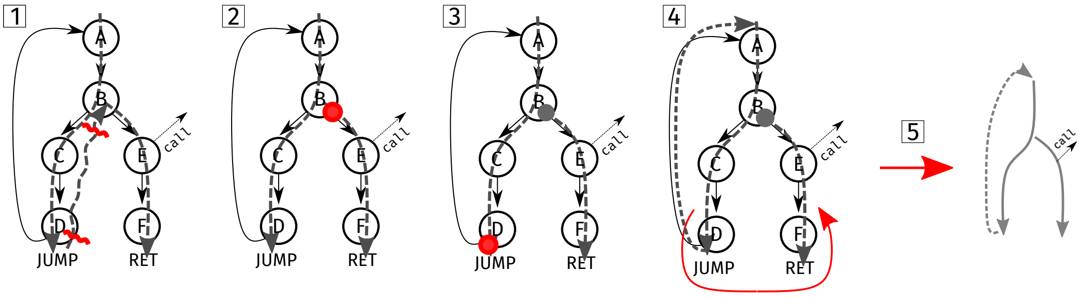

1
A
B
C
D
E
F
call
JUMP
RET
2
3
4
5
call

FIGURE 5.5: The working flow of trace stitching.

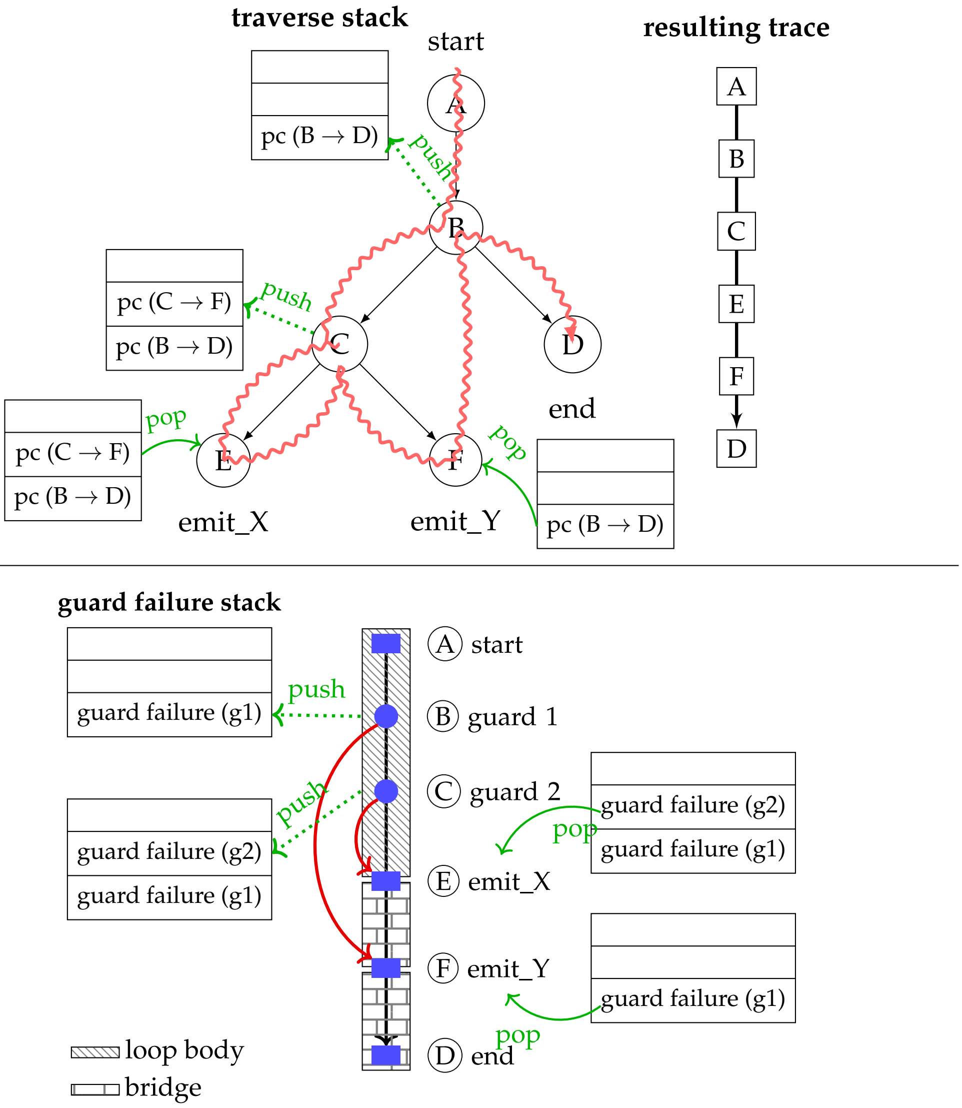
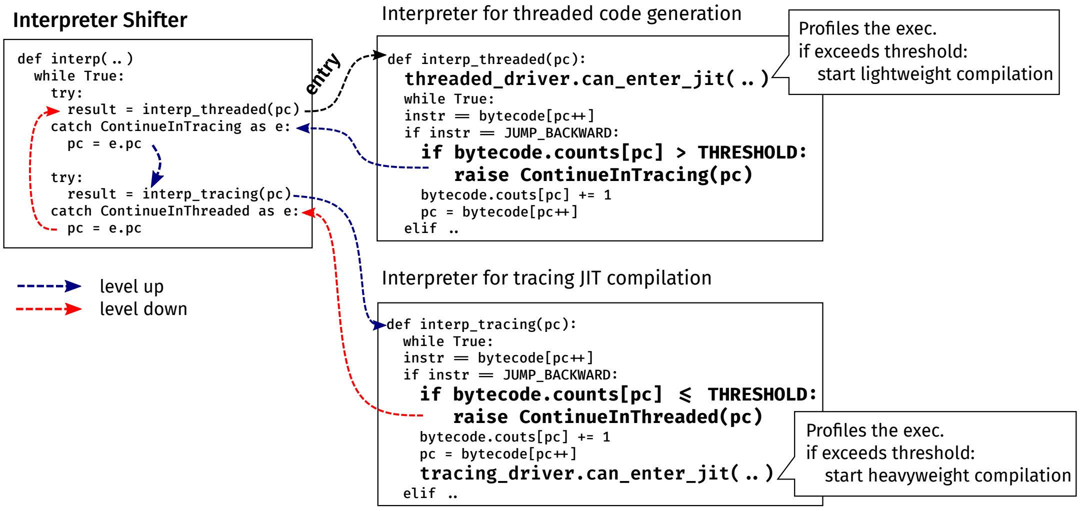

traverse stack
start
resulting trace
pc (B → D)
push
A
B
C
D
E
F
pc (C → F)
pc (B → D)
push
end
pop
pop
emit_X
emit_Y
guard failure stack
guard failure (g1)
push
A start
B guard 1
C guard 2
push
guard failure (g2)
guard failure (g1)
pop
E emit_X
F emit_Y
D end
pop
loop body
bridge

FIGURE 5.6: Overview of trace-stitching. This shows how we resolve the relations between guard failures and bridges.


Interpreter Shifter
entry
Interpreter for threaded code generation
Profiles the exec.
if exceeds threshold:
start lightweight compilation
Interpreter for tracing JIT compilation
Profiles the exec.
if exceeds threshold:
start heavyweight compilation
level up
level down

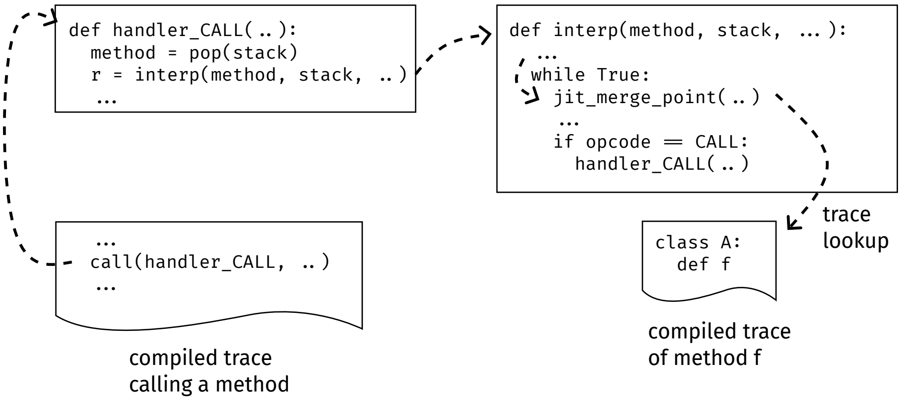
```
def interp( .. )
  while True:
    try:
      result = interp_threaded(pc)
    catch ContinueInTracing as e:
      pc = e.pc

    try:
      result = interp_tracing(pc)
    catch ContinueInThreaded as e:
      pc = e.pc
```

```
def interp_threaded(pc):
  threaded_driver.can_enter_jit( .. )
  while True:
  instr == bytecode[pc++]
  if instr == JUMP_BACKWARD:
    if bytecode.counts[pc] > THRESHOLD:
      raise ContinueInTracing(pc)
    bytecode.couts[pc] += 1
    pc = bytecode[pc++]
  elif ..
```

```
def interp_tracing(pc):
  while True:
  instr == bytecode[pc++]
  if instr == JUMP_BACKWARD:
    if bytecode.counts[pc] ⩽ THRESHOLD:
      raise ContinueInThreaded(pc)
    bytecode.couts[pc] += 1
    pc = bytecode[pc++]
    tracing_driver.can_enter_jit( .. )
  elif ..
```

FIGURE 5.7: Overview of the runtime techniques for multilevel compilation. The interpreter shifting defines the strategy for transitioning between compilation levels. Note that each interpreter shown on the right-hand side corresponds to each compilation level. For each interpreter, the drivers profile the execution and starts JIT compilation when the threshold is exceeded.


trace
lookup
compiled trace
calling a method
compiled trace
of method f

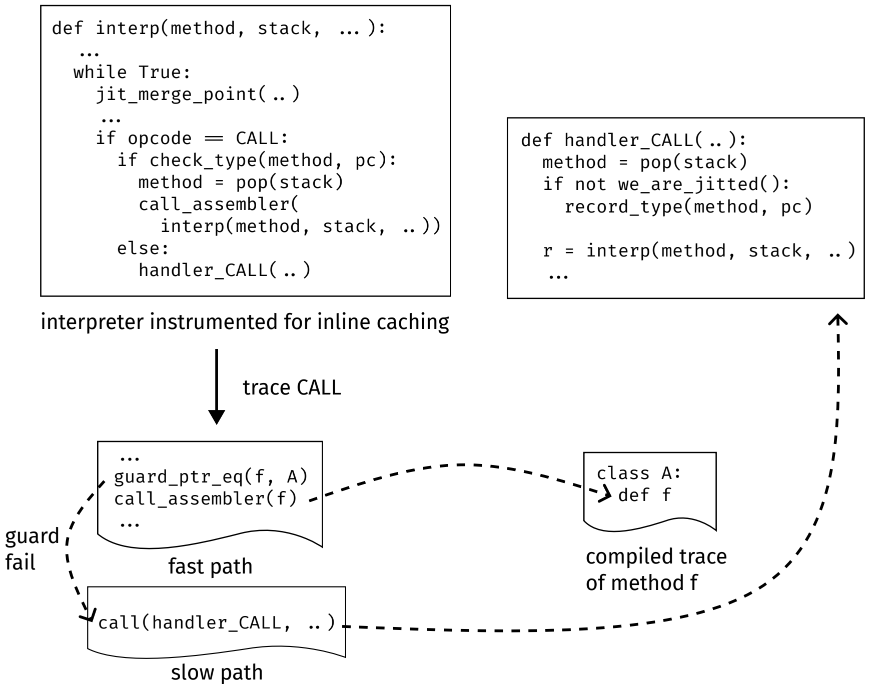
```
def handler_CALL( .. ):
  method = pop(stack)
  r = interp(method, stack, .. )
  ...
```

```
def interp(method, stack, ... ):
  ...
  while True:
    jit_merge_point( .. )
    ...
    if opcode == CALL:
      handler_CALL( .. )
```

```
...
call(handler_CALL, .. )
...
```

```
class A:
  def f
```

FIGURE 5.8: Calling a compiled method w/o inline caching.



```
def interp(method, stack, ...):
  ...
  while True:
    jit_merge_point(..)
    ...
    if opcode == CALL:
      if check_type(method, pc):
        method = pop(stack)
        call_assembler(
          interp(method, stack, ..))
      else:
        handler_CALL(..)
```

```
def handler_CALL(..):
  method = pop(stack)
  if not we_are_jitted():
    record_type(method, pc)

  r = interp(method, stack, ..)
  ...
```

```
 ...
guard_ptr_eq(f, A)
call_assembler(f)
 ...
```

```
class A:
  def f
```

```
call(handler_CALL, ..)
```

interpreter instrumented for inline caching
trace CALL
guard fail
fast path
compiled trace of method f
slow path

FIGURE 5.9: Overview of inline caching in threaded code generation.

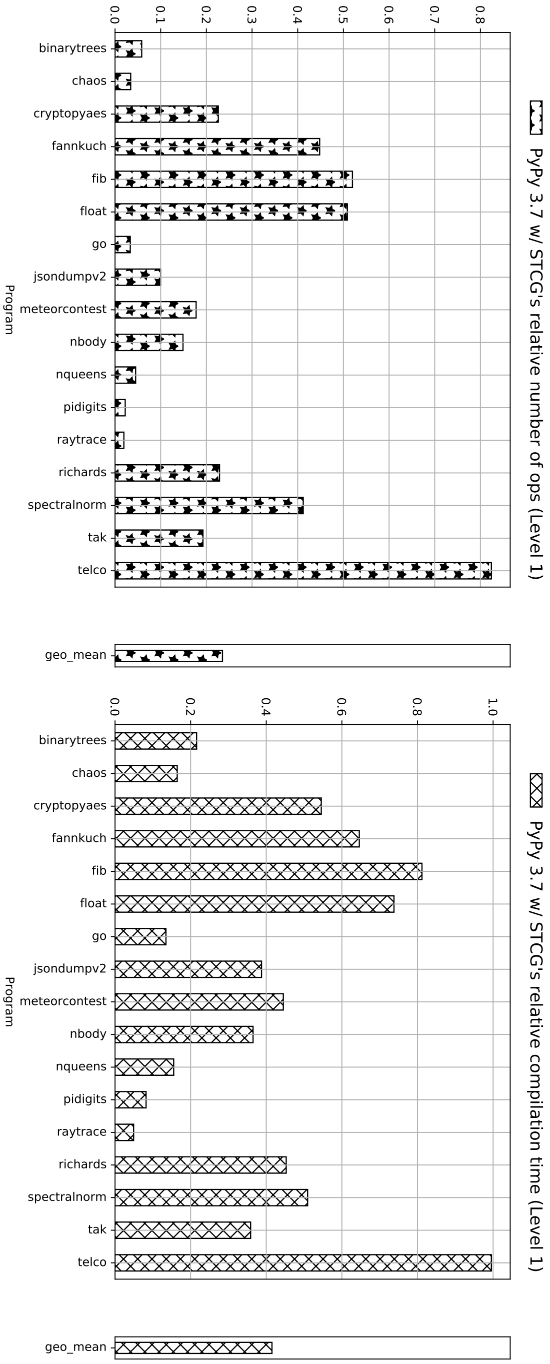


FIGURE 5.10: The results of the size of traces to compile and compilation time including tracing. In all results the Y-axis means PyPy 3.7 with our simulated threaded code generation (STCG)'s relative value to PyPy 3.7–7.3.5's tracing JIT compiler. The X-axis stands for the name of every program. The left-hand side shows the relative trace size, and the right-hand size is the relative compilation time. Lower is better.

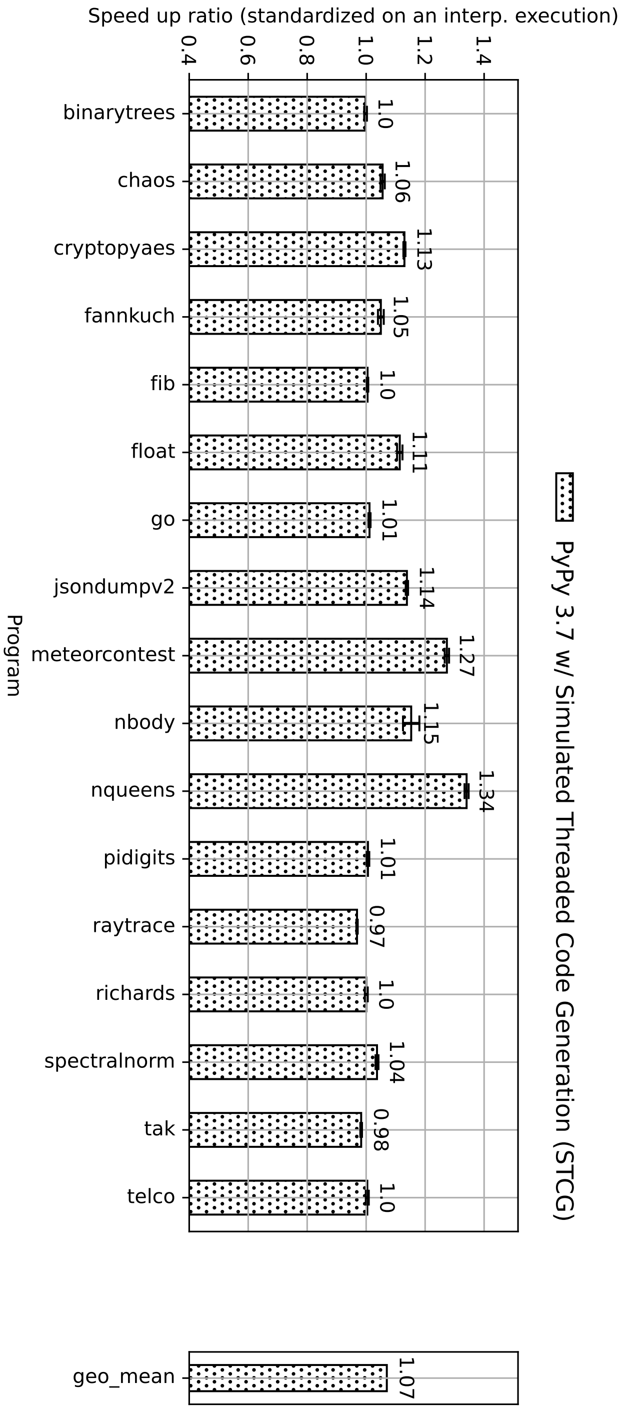


FIGURE 5.11: The results of a preliminary benchmark experiment. In all results the Y-axis means speed up ratio of the threaded code generation compared with the interpreter-only execution, and the X-axis stands for the name of every program. The error bars mean standard deviations. Higher is better.

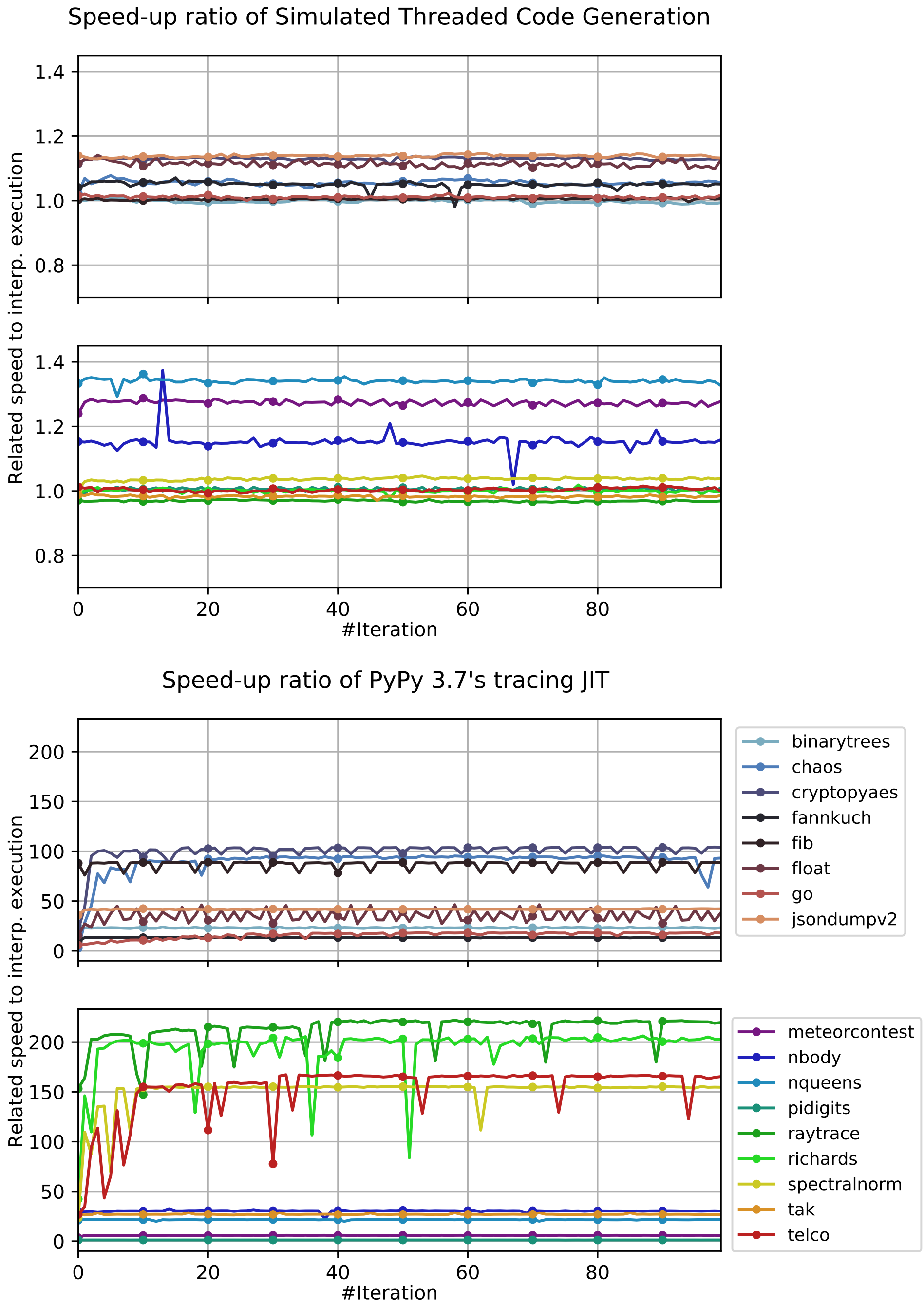


FIGURE 5.12: Speed-up ratio of STCG (left-hand side) and PyPy's tracing JIT compiler (right-hand side) related to the interpreter. They are executed on PyPy's original micro benchmark suite plus our original ones. X-axis and Y-axis mean every iteration and speed-up ratio standardized to interpreter execution, respectively. Dots are plotted every five iterations.

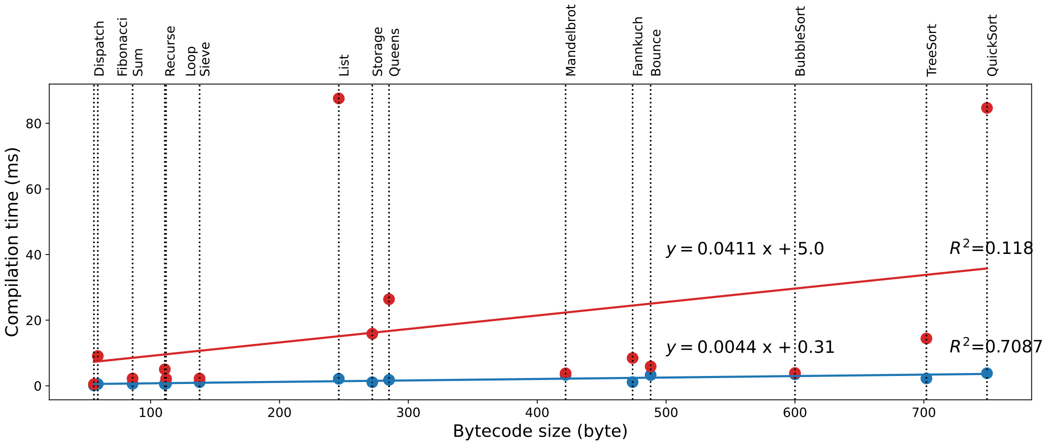


FIGURE 5.13: Compilation times and code sizes on micro-benchmark programs. The x-axis and y-axis mean a bytecode size (byte) and consumed compilation time (ms), respectively.

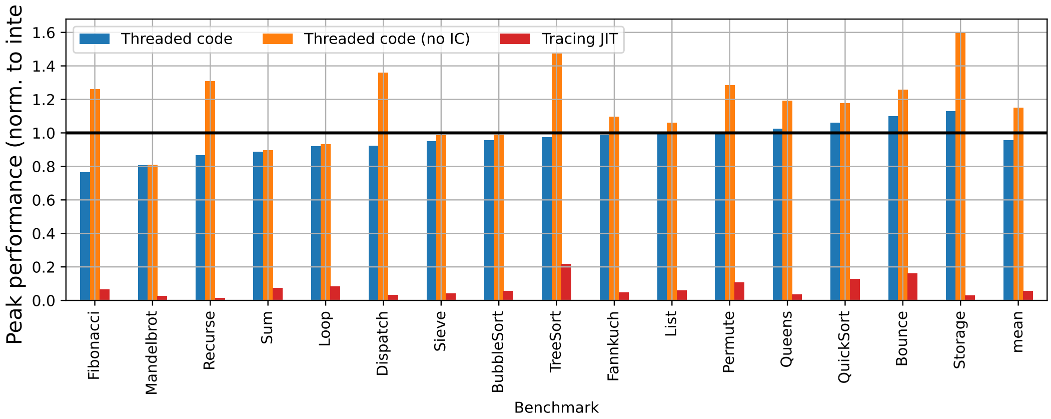


FIGURE 5.14: Peak performance of threaded code generation, tracing JIT and interpreter execution. Relative elapsed times normalized to interpreter execution. The blue and red lines mean threaded code generation and tracing JIT, respectively. Lower is better.

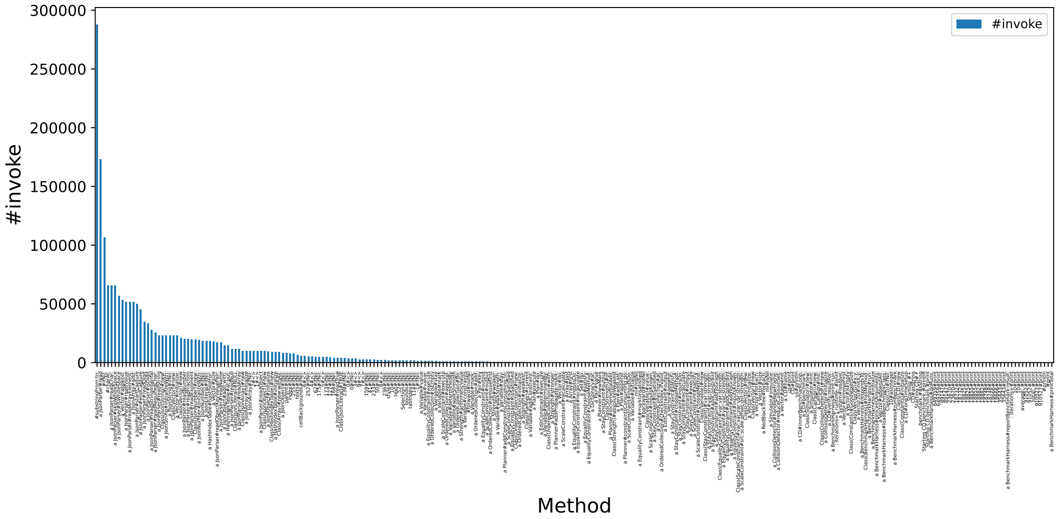


FIGURE 5.15: Number of method invocations in the simulated real-world workload application.

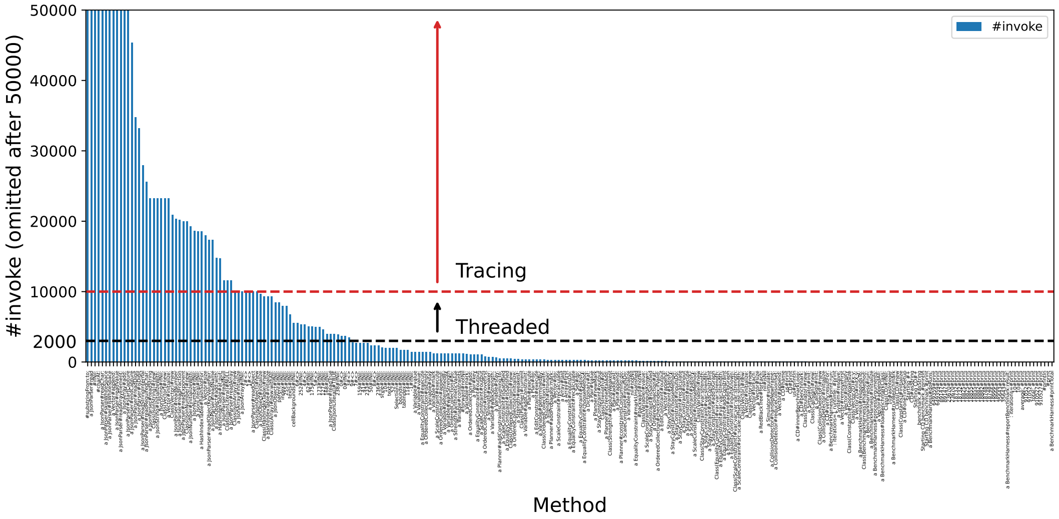


FIGURE 5.16: Visualization of which methods compile at which level by defined thresholds using Figure 5.15.

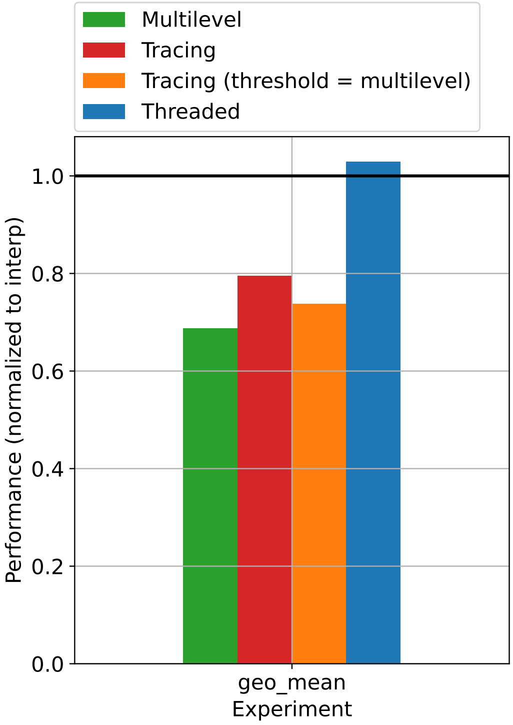


FIGURE 5.17: Results of the experiment in the simulated real-world workload application.

Chapter 6

Conclusion

A meta-tracing JIT compiler makes it easier to write a dynamic language without having to create a JIT compiler from scratch. This technique is achieved by generating a VM with a tracing JIT compiler from an interpreter definition. There is a wide range of JIT compilation policies, such as method based, trace based, region based, and so forth, but there is no clear winner between them. To further improve the performance of the JIT compiler, it is necessary to implement a multi-role compilation method that combines multiple JIT compilation scopes and optimization levels.

The goal of present dissertation has been to realize a multi-role meta-tracing compilation. To achieve this goal, the author had to show the possibility of realizing a mechanism that can change a compilation unit and an optimization level in a meta-tracing JIT compiler framework. To achieve this goal, this dissertation followed two steps: adding (1) a new compilation policy with a different compilation scope (multi-scope) and (2) a new compilation/optimization behavior (multi-level) to a meta-tracing JIT compiler.

This dissertation first showed the mechanism to mix method- and trace-based heavyweight compilers in a meta-tracing JIT compiler framework. The proof-of-concept implementation based on the MinCaml compiler, namely BacCaml, was also proposed. The evaluation of BacCaml found that there existed a program that ran faster with multi-scope JIT compilation by up to 59% against a single JIT compilation policy.

The second part of this dissertation proposed Multilevel RPython, which mixes method-based lightweight and trace-based heavyweight JIT compilations in a real-world meta-tracing JIT compiler. This approach is achieved by inserting new hint instructions into an interpreter, where instructions give the compiler that does the meta-tracing a “different behavior” instead of building a compiler from scratch. The reason for mixing the method-based lightweight compilation was to improve the overall performance in conjunction with trace-based heavyweight compilation by applying heavyweight compilation to hot spots and lightweight compilation to warm spots. In the evaluation and experiment of the Smalltalk implementation subset called PySOM, it was confirmed that the overall performance was 14% faster with multilevel execution than tracing JIT-only execution.

Considering the applicability of multi-role compilation to another meta-JIT compiler framework called Truffle/Graal is left for future work. Truffle/Graal specializes an AST interpreter written in the Truffle framework. For achieving multi-role compilation for the AST-specializing system, it could be done by generating another node for trace-based compilation. The generated AST nodes are instrumented by annotations to limit the compilation scope to being linear.

In addition, the author plans to realize a method-based heavyweight JIT compilation on Multilevel RPython in future work. The limitation of the current version of Multilevel RPython is not having another choice of method-based compilation, so it would be worth creating this on RPython. This plan would be extended to realize a new meta-compiler with several optimization levels and different compilation units. It can be possible to investigate if adaptive compilation using interpreters is interesting in the future, which adaptively transitions between those compiler/optimization levels with various characteristics. This is needed to implement a profiler and more dedicated strategy to switch optimization levels in an interpreter. Finally, the author would like to introduce this research into PyPy to verify whether this research would also be effective for applications with workload characteristics found in real software development. We hope this helps all developers in the field of software development.

Bibliography

Anderson, Kenneth R. and Duane Rettig (Oct. 1994). "Performing Lisp Analysis of the FANNKUCH Benchmark". In: *SIGPLAN Lisp Pointers* VII.4, pp. 2–12. ISSN: 1045-3563. DOI: 10.1145/382109.382124. URL: https://doi.org/10.1145/382109.382124.

Arnold, Matthew et al. (2000). "Adaptive Optimization in the Jalapeño JVM". In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. Minneapolis, Minnesota, USA: Association for Computing Machinery, pp. 47–65. ISBN: 158113200X. DOI: 10.1145/353171.353175. URL: https://doi.org/10.1145/353171.353175.

Aycock, John (June 2003). "A Brief History of Just-in-Time". In: *ACM Comput. Surv.* 35.2, pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

Bala, Vasanth, Evelyn Duesterwald, and Sanjeev Banerjia (2000). "Dynamo: a Transparent Dynamic Optimization System". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ISBN: 1-58113-199-2. DOI: 10.1145/349299.349303. eprint: 1003.4074.

Barati, Saam (2022). *A design overview of JavaScriptCore's DFG IR*. Presentation at ICOOOLPS'22: Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems.

Bauman, Spenser et al. (2015). "Pycket: A Tracing JIT for a Functional Language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: ACM, pp. 22–34. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784740.

Bebenita, Michael et al. (2010). "SPUR: A Trace-based JIT Compiler for CIL". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, pp. 708–725. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869517.

Bell, James R. (June 1973). "Threaded Code". In: *Commun. ACM* 16.6, pp. 370–372. ISSN: 0001-0782. DOI: 10.1145/362248.362270.

Bellard, Fabrice (2005). "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, p. 41.

Blackburn, Stephen M. et al. (2006). "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, pp. 169–190. ISBN: 1595933484. DOI: 10.1145/1167473.1167488. URL: https://doi.org/10.1145/1167473.1167488.

Bolz, Carl Friedrich, Lukas Diekmann, and Laurence Tratt (2013). “Storage Strategies for Collections in Dynamically Typed Languages”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: Association for Computing Machinery, pp. 167–182. ISBN: 9781450323741. DOI: 10.1145/2509136.2509531. URL: https://doi.org/10.1145/2509136.2509531.

Bolz, Carl Friedrich et al. (2009). “Tracing the Meta-level: PyPy’s Tracing JIT Compiler”. In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. Genova, Italy: ACM, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.

Bolz, Carl Friedrich et al. (2011a). “Allocation Removal by Partial Evaluation in a Tracing JIT”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’11. Austin, Texas, USA: ACM, pp. 43–52. ISBN: 978-1-4503-0485-6. DOI: 10.1145/1929501.1929508.

– (2011b). “Runtime Feedback in a Meta-tracing JIT for Efficient Dynamic Languages”. In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS ’11. Lancaster, United Kingdom: ACM, 9:1–9:8. ISBN: 978-1-4503-0894-6. DOI: 10.1145/2069172.2069181.

Budimlic, Zoran and Ken Kennedy (June 1997). “Optimizing Java: theory and practice”. en. In: *Concurrency Practice and Experience* 9.6, pp. 445–463. ISSN: 1040-3108, 1096-9128. DOI: 10.1002/(sici)1096-9128(199706)9:6<445::aid-cpe301>3.0.co;2-l.

Casey, Kevin et al. (2003). “Towards Superinstructions for Java Interpreters”. In: *Software and Compilers for Embedded Systems*. Springer Berlin Heidelberg, pp. 329–343. DOI: 10.1007/978-3-540-39920-9_23.

Chevalier-Boisvert, Maxime and Marc Feeley (2015). “Simple and Effective Type Check Removal through Lazy Basic Block Versioning”. In: *29th European Conference on Object-Oriented Programming (ECOOP15)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 101–123. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.101.

– (2016). “Interprocedural Type Specialization of JavaScript Programs Without Type Analysis”. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 7:1–7:24. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.7.

Chevalier-Boisvert, Maxime et al. (Oct. 2021). “YJIT: a basic block versioning JIT compiler for CRuby”. In: *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: Association for Computing Machinery, pp. 25–32. ISBN: 9781450391092. DOI: 10.1145/3486606.3486781.

Choi, Jong-Deok et al. (1999b). "Escape Analysis for Java". In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '99. Denver, Colorado, USA: Association for Computing Machinery, pp. 1–19. ISBN: 1581132387. DOI: 10.1145/320384.320386. URL: https://doi.org/10.1145/320384.320386.

Choi, Jong-Deok et al. (Oct. 1999a). "Escape analysis for Java". In: *SIGPLAN Not.* 34.10, pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/320385.320386.

Coffin, Eric et al. (2020). "MicroJIT: A Case for Templated Just-in-Time Compilation in Constrained Environments". In: *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*. CASCON '20. Toronto, Ontario, Canada: IBM Corp., pp. 179–188.

Deutsch, L. Peter and Allan M. Schiffman (1984). "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: Association for Computing Machinery, pp. 297–302. ISBN: 0897911253. DOI: 10.1145/800017.800542.

Eclipse Foundation (2017). URL: https://www.eclipse.org/openj9/.

– (2019). *Eclipse OMR^{TM} Cross platform components for building reliable, high performance language runtimes*. URL: https://www.eclipse.org/omr/.

Ertl, M Anton and David Gregg (2003). "Implementation Issues for Superinstructions in Gforth". In: *Proceedings of EuroForth*.

Ertl, M. Anton and David Gregg (2003). "The Structure and Performance of Efficient Interpreters." In: *Journal of Instruction-level Parallelism* 5.

Ertl, M Anton et al. (Mar. 2002). "Vmgen: a generator of efficient virtual machine interpreters". In: *Software: practice & experience* 32.3, pp. 265–294. ISSN: 0038-0644. DOI: 10.1002/spe.434.

Felgentreff, Tim et al. (2016). "How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST '16. Prague, Czech Republic: ACM, 21:1–21:10. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991062.

Gal, Andreas, Christian W. Probst, and Michael Franz (2006). "HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: Association for Computing Machinery, pp. 144–153. ISBN: 1595933328. DOI: 10.1145/1134760.1134780.

Gal, Andreas et al. (2009). "Trace-Based Just-in-Time Type Specialization for Dynamic Languages". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, pp. 465–478. ISBN: 9781605583921. DOI: 10.1145/1542476.1542528. URL: https://doi.org/10.1145/1542476.1542528.

Gaynor, Alex et al. (2013). *A high performance ruby, written in RPython*. URL: http://docs.topazruby.com/en/latest/.

Google (2015a). *Digging into the TurboFan JIT*. URL: https://v8.dev/blog/turbofan-jit.

Google (2015b). *Googles High-performance Open Source JavaScript and WebAssembly Engine*. URL: https://v8.dev/.

– (2016). *Firing up the Ignition interpreter*. URL: https://v8.dev/blog/ignition-interpreter.

Hank, Richard E., Wen-Mei W. Hwu, and B. Ramakrishna Rau (1995). "Region-Based Compilation: An Introduction and Motivation". In: *Proceedings of the 28th Annual International Symposium on Microarchitecture*. MICRO 28. Ann Arbor, Michigan, USA: IEEE Computer Society Press, pp. 158–168. ISBN: 0818673494.

Haupt, Michael et al. (2010). "The SOM Family: Virtual Machines for Teaching and Research". In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '10. Bilkent, Ankara, Turkey: ACM, pp. 18–22. ISBN: 978-1-60558-820-9. DOI: 10.1145/1822090.1822098.

Hayashizaki, Hiroshige et al. (2011). "Improving the Performance of Trace-based Systems by False Loop Filtering". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, pp. 405–418. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950412.

Hong, P. Joseph (Oct. 1992). "Threaded Code Designs for Forth Interpreters". In: *SIGFORTH Newsl.* 4.2, pp. 11–16. ISSN: 1047-4544. DOI: 10.1145/146559.146561.

Huang, Ruochen, Hidehiko Masuhara, and Tomoyuki Aotani (2016). "Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler". In: *International Symposium on Trends in Functional Programming*. Springer, pp. 44–58.

Hölzle, Urs, Craig Chambers, and David Ungar (1991). "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches". In: *ECOOP'91 European Conference on Object-Oriented Programming*. Ed. by Pierre America. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 21–38. ISBN: 978-3-540-47537-8.

Iliasov, Alex (Aug. 2003). "Templates-Based Portable Just-in-Time Compiler". In: *SIGPLAN Not.* 38.8, pp. 37–43. ISSN: 0362-1340. DOI: 10.1145/944579.944588. URL: https://doi.org/10.1145/944579.944588.

Inoue, Hiroshi et al. (2011). "A trace-based Java JIT Compiler Retrofitted from a Method-based Compiler". In: pp. 246–256. ISBN: 9781612843551. DOI: 10.1109/CGO.2011.5764692.

Ishizaki, Kazuaki et al. (June 1999). "Design, implementation, and evaluation of optimizations in a just-in-time compiler". In: *Proceedings of the ACM 1999 conference on Java Grande*. JAVA '99. San Francisco, California, USA: Association for Computing Machinery, pp. 119–128. ISBN: 9781581131611. DOI: 10.1145/304065.304111.

Izawa, Yusuke and Hidehiko Masuhara (2020). "Amalgamating Different JIT Compilations in a Meta-Tracing JIT Compiler Framework". In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS '20. Virtual, USA: Association for Computing Machinery, pp. 1–15. ISBN: 9781450381758. DOI: 10.1145/3426422.3426977.

Izawa, Yusuke et al. (2022). "Threaded Code Generation with a Meta-Tracing JIT Compiler". In: *Journal of Object Technology* 21.2, a1. DOI: `10.5381/jot.2022.21.2.a1`.

Kotzmann, Thomas et al. (May 2008). "Design of the Java HotSpot™ Client Compiler for Java 6". In: *ACM Trans. Archit. Code Optim.* 5.1. ISSN: 1544-3566. DOI: `10.1145/1369396.1370017`. URL: `https://doi.org/10.1145/1369396.1370017`.

Krylov, Georgiy et al. (2021). "Ahead-of-Time Compilation in Eclipse OMR on Example of WebAssembly". In: *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*. CASCON '21. Toronto, Canada: IBM Corp., pp. 237–243.

Marr, Stefan (2018). *ReBench: Execute and Document Benchmarks Reproducibly*. Version 1.0. DOI: `10.5281/zenodo.1311762`.

Marr, Stefan, Benoit Daloze, and Hanspeter Mössenböck (Nov. 2016). "Cross-Language Compiler Benchmarking: Are We Fast Yet?" In: *SIGPLAN Not.* 52.2, pp. 120–131. ISSN: 0362-1340. DOI: `10.1145/3093334.2989232`. URL: `https://doi.org/10.1145/3093334.2989232`.

Miranda, Eliot (Nov. 1999). "Context Management in VisualWorks 5i". In: *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*. Denver, CO. URL: `http://www.esug.org/data/Articles/misc/oopsla99-contexts.pdf`.

Nanjekye, Joannah, David Bremner, and Aleksandar Micic (2021). "Eclipse OMR Garbage Collection for Tracing JIT-Based Virtual Machines". In: *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*. CASCON '21. Toronto, Canada: IBM Corp., pp. 244–249.

Niephaus, Fabio, Tim Felgentreff, and Robert Hirschfeld (2018). "GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework". In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '18. Amsterdam, Netherlands: ACM, pp. 30–35. ISBN: 978-1-4503-5804-0. DOI: `10.1145/3242947.3242948`.

– (2019). "GraalSqueak: Toward a Smalltalk-based Tooling Platform for Polyglot Programming". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR '19. Athens, Greece: ACM, pp. 14–26. ISBN: 978-1-4503-6977-0. DOI: `10.1145/3357390.3361024`.

Ocacle Lab. (2019). *A ECMAScript 2019 compliant Javascript implementation built on GraalVM. With polyglot language interoperability support.* URL: `https://github.com/graalvm/graaljs`.

Oracle Lab. (2013). *A high performance implementation of the Ruby programming language*. URL: `https://github.com/oracle/truffleruby`.

– (2015). *A high-performance implementation of the R programming language, built on GraalVM*. URL: `https://github.com/oracle/fastr`.

– (2022). *Host Compilation for Interpreter Java code*. URL: `https://github.com/oracle/graal/blob/master/truffle/docs/HostCompilation.md#host-inlining`.

Ottoni, Guilherme (June 2018). "HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack". In: *SIGPLAN Not.* 53.4, pp. 151–165. ISSN: 0362-1340. DOI: `10.1145/3296979.3192374`. URL: `https://doi.org/10.1145/3296979.3192374`.

Paleczny, Michael, Christopher Vick, and Cliff Click (2001). "The Java Hotspot™ Server Compiler". In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM '01. Monterey, California: USENIX Association, p. 1.

Pall, Mike (2005). *A Just-in-time Compiler for Lua Programming Language*. URL: `http://luajit.org/index.html`.

Piumarta, Ian and Fabio Riccardi (1998). "Optimizing Direct Threaded Code by Selective Inlining". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 291–300. ISBN: 0897919874. DOI: `10.1145/277650.277743`.

Pizlo, Fillip (2014). *Introducing the WebKit FTL JIT*. URL: `https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/`.

– (2020). *Speculation in JavaScriptCore*. URL: `https://webkit.org/blog/10308/speculation-in-javascriptcore/`.

Rigger, Manuel et al. (2016). "Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle". In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. VMIL '16. Amsterdam, Netherlands: ACM, pp. 6–15. ISBN: 978-1-4503-4645-0. DOI: `10.1145/2998415.2998416`.

Rigo, Armin and Samuele Pedroni (2006). "PyPys Approach to Virtual Machine Construction". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, pp. 944–953. ISBN: 159593491X. DOI: `10.1145/1176617.1176753`.

Steele, Guy Lewis (1977). "Debunking the Expensive Procedure Call Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO". In: *Proceedings of the 1977 Annual Conference*. ACM '77. Seattle, Washington: Association for Computing Machinery, pp. 153–162. ISBN: 9781450339216. DOI: `10.1145/800179.810196`. URL: `https://doi.org/10.1145/800179.810196`.

Suganuma, Toshio, Toshiaki Yasue, and Toshio Nakatani (Jan. 2006). "A Region-Based Compilation Technique for Dynamic Compilers". In: vol. 28. 1. New York, NY, USA: Association for Computing Machinery, pp. 134–174. DOI: `10.1145/1111596.1111600`.

Sumii, Eijiro (2005). "MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language". In: pp. 27–38. ISBN: 1595930671. DOI: `10.1145/1085114.1085122`.

Team, PyPy JS (2015). *PyPy compiled into JavaScript*. URL: `https://github.com/pypyjs/pypyjs`.

"The IBM J9 Java Virtual Machine for Java 6" (2009). In: *Pro IBMő WebSphereő Application Server 7 Internals*. Berkeley, CA: Apress, pp. 15–34. ISBN: 978-1-4302-1959-0. DOI: 10.1007/978-1-4302-1959-0_2. URL: https://doi.org/10.1007/978-1-4302-1959-0_2.

Ungar, David and Randall B. Smith (1987). "Self: The Power of Simplicity". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '87. Orlando, Florida, USA: Association for Computing Machinery, pp. 227–242. ISBN: 0897912470. DOI: 10.1145/38765.38828. URL: https://doi.org/10.1145/38765.38828.

Venners, Bill (1998). "The java virtual machine". In: *Java and the Java virtual machine: definition, verification, validation*.

Wimmer, Christian et al. (Jan. 2013). "Maxine: An Approachable Virtual Machine for, and in, Java". In: *ACM Trans. Archit. Code Optim.* 9.4. ISSN: 1544-3566. DOI: 10.1145/2400682.2400689. URL: https://doi.org/10.1145/2400682.2400689.

Wimmer, Christian et al. (Oct. 2019). "Initialize Once, Start Fast: Application Initialization at Build Time". In: *Proc. ACM Program. Lang.* 3.OOPSLA. DOI: 10.1145/3360610. URL: https://doi.org/10.1145/3360610.

Würthinger, Thomas et al. (2012). "Self-Optimizing AST Interpreters". In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS '12. Tucson, Arizona, USA: Association for Computing Machinery, pp. 73–82. ISBN: 9781450315647. DOI: 10.1145/2384577.2384587.

Würthinger, Thomas et al. (2017). "Practical Partial Evaluation for High-performance Dynamic Language Runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, pp. 662–676. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062381.

Xu, Haoran and Fredrik Kjolstad (Oct. 2021). "Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode". In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: 10.1145/3485513. URL: https://doi.org/10.1145/3485513.