UNIVERSITY OF CALIFORNIA,
IRVINE

Trace-Based Compilation and Optimization in Meta-Circular Virtual Execution Environments

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Michael Alexander Bebenita

Dissertation Committee:
Professor Michael Franz, Chair
Professor Brian Demsky
Professor Ian Harris
Professor Tony Givargis
Dr. Andreas Gal

2011

The dissertation of Michael Alexander Bebenita
is approved and is acceptable in quality and form for
publication on microfilm and in digital formats:

_____

_____

_____

Committee Chair

University of California, Irvine
2011

DEDICATION

TABLE OF CONTENTS

Page

LIST OF FIGURES vii

LIST OF TABLES ix

ACKNOWLEDGMENTS x

CURRICULUM VITAE xi

ABSTRACT OF THE DISSERTATION xv

1 Introduction 1

2 Background 3
- 2.1 Virtual Machines . . . 3
  - 2.1.1 Short History of Just-in-Time Compilation . . . 4
  - 2.1.2 The Java Virtual Machine . . . 6
  - 2.1.3 Interpreters . . . 7
  - 2.1.4 Just-in-Time Compilers . . . 8
- 2.2 Trace Optimizations . . . 10
  - 2.2.1 Trace Cache . . . 10
  - 2.2.2 Trace Scheduling . . . 11
  - 2.2.3 Dynamic Binary Translators . . . 12
  - 2.2.4 Summary . . . 12

3 Evolution of Trace Regions 13
- 3.1 A Motivating Example . . . 13
- 3.2 Representing Partial Programs using Traces . . . 16
  - 3.2.1 Trace Trees . . . 18
  - 3.2.2 Nested Trace Trees . . . 19
  - 3.2.3 Trace Regions . . . 22

4 J^{2}VM 24
- 4.1 J^{2}VM . . . 24
  - 4.1.1 The J^{2}VM Interpreter . . . 26
- 4.2 Trace-Based Compilation . . . 31
  - 4.2.1 Trace Anchor Detection . . . 32
  - 4.2.2 Trace Tree Recording . . . 33
  - 4.2.3 Trace Specific Compiler Optimizations . . . 33

4.2.4 Code Loading, Linking & Execution . . . 36

5 Maxpath **44**
5.1 Introduction . . . 44
5.2 Maxine VM . . . 45
5.2.1 Meta-Circular Design . . . 45
5.2.2 Bootstrapping & Execution . . . 46
5.2.3 Baseline & Optimizing Compilers . . . 47
5.2.4 Stack Frame Layouts & Adapter Frames . . . 48
5.2.5 Garbage Collection & Safepointing . . . 51
5.3 Maxpath . . . 52
5.3.1 Discovering Hot Program Regions . . . 53
5.3.2 Trace Recording . . . 54
5.3.3 Heuristics . . . 59
5.3.4 Trace Sampling . . . 61
5.3.5 Trace Region Compilation & Execution . . . 68
5.4 Conclusion . . . 72

6 Trace Driven Incremental SSA Form Construction **75**
6.1 State Vectors . . . 75
6.1.1 SSA Values . . . 76
6.2 SSA Control Flow Graph . . . 76
6.2.1 SSA Value Forwarding . . . 77
6.2.2 Basic Block Construction & State Vector Linking . . . 78
6.3 Incremental SSA Form Construction . . . 79
6.3.1 Dynamic Control Flow Graphs . . . 80
6.3.2 State Maps & Equivalence Relations . . . 81
6.4 Implementation . . . 84
6.4.1 C1X Compiler Architecture . . . 85
6.4.2 C1X Tracing Architecture . . . 85

7 Applications of Trace-Based Compilers **91**
7.1 Tracing in Dynamic Languages . . . 91
7.1.1 Static Type Inference . . . 92
7.1.2 Deferred Runtime Type Inference . . . 93
7.2 Trace-Based Compilers for the CLR . . . 96
7.2.1 Optimizations . . . 98

8 Related Work **106**

9 Evaluation **108**
9.1 Benchmarks . . . 108
9.2 Execution Modes . . . 110
9.3 Performance . . . 112
9.3.1 Java Grande . . . 112
9.3.2 Spec 2008 . . . 113
9.3.3 DaCapo . . . 113
9.3.4 Hotspot Comparison . . . 113

9.4 Memory Consumption 117
9.4.1 Profiling Instrumentation 117
9.4.2 Code Size 120
9.5 Tracer Recording Metrics 121
9.5.1 Trace Region Metrics 121

10 Conclusions **128**
10.1 Acknowledgements 129

Bibliography **130**

LIST OF FIGURES

Page

2.1 Time, space and latency tradeoffs of compilers and interpreters. 4
2.2 Java Bytecode 5
2.3 A sequence of bytecode operations for the Java statement `y = 2 + x`. . . . 7
2.4 Lightweight vs. Optimizing JIT Compilers 10

3.1 Speculative method inlining vs. trace-based compilation. 15
3.2 Linear trace. 16
3.3 Stitching of overlapping linear traces. 17
3.4 Growth of a trace tree, with excessive tail duplication. 19
3.5 Pathological cases for trace tree coverage. 19
3.6 Hot loop pair. 20
3.7 Nested trace trees. 21
3.8 Growth of a trace region. 23

4.1 J^2VM architecture. 27
4.2 Performance of J^2VM on Java Grande section 2 and 3. 29
4.3 Proportion of time spent in compiled code for sections 2 and 3. 29
4.4 Simulated performance of J^2VM in a mixed-mode VM. 30
4.5 Trace-based compilation architecture in J^2VM. 31
4.6 JVML reuse of local variables. 33
4.7 Partially invariant code motion example. 36
4.8 Using the Java reflection API to access private class members. 38
4.9 Compiled trace tree. 40
4.10 Executing dynamically generated code through the Java Native Interface. . 43

5.1 Maxine VM components 46
5.2 Conventional VM design vs. Meta-Circular VM design. 46
5.3 Bootstrapping Maxine 48
5.4 Maxine execution phases. 49
5.5 Contrast between the baseline and optimizing compiler. 49
5.6 Stack frame layouts. 50
5.7 Adapter frames. 51
5.8 Execution stack with baseline, trace regions and optimized methods interleaved. 51
5.9 Interpreter vs. just-in-time based trace recorder. 56
5.10 Switching between instrumented methods. 57
5.11 Implementation of `visitAnchor` 57
5.12 Implementation of `visitBytecode` instrumentation in Java. 57
5.13 Tracer scopes. 59

5.14 A simple loop with two equally hot paths. . . . 60
5.15 Relationship between Baseline and Maxpath in terms of performance. . . . 62
5.16 Application execution life cycle. . . . 63
5.17 Trace region formation and execution life cycle. . . . 63
5.18 Decision diagram for the `visitAnchor` message. . . . 64
5.19 Trace region adapter frame. . . . 69
5.20 Three transfer types: Inlined, Out-of-Line and Slow Transfers. . . . 70
5.21 Bubble sort algorithm in Java and JVML Bytecode . . . 73
5.22 Transfer of Control . . . 74

6.1 Four examples of Java bytecode that operate on SSA state vectors. . . . 76
6.2 Merging of entry and exit state vectors. . . . 77
6.3 Construction of a dynamic control flow graph using equivalence relations. . . 84
6.4 Architecture of the C1X compiler. . . . 86
6.5 State propagation in the Method Graph Builder. . . . 87
6.6 Order in which the Method Graph Builder processes basic blocks. . . . 88
6.7 Order in which the Trace Graph Builder processes basic blocks. . . . 88
6.8 Tail duplication in the Trace Graph Builder. . . . 89

7.1 A simple accumulation loop in JavaScript. . . . 93
7.2 JavaScript VM layered on top of a trace-based JVM . . . 94
7.3 Partial implementation of `JsValue` in Java. . . . 100
7.4 Statically typed JVML code generated for JavaScript code. . . . 101
7.5 Code in figure 7.4 on page 101 with only one inlined `add` method call. . . . 101
7.6 Unoptimized trace recorded for the program in figure 7.4 on page 101. . . . 102
7.7 Optimized trace. . . . 103
7.8 Optimized trace. . . . 104
7.9 Architecture of SPUR system. . . . 104
7.10 Java vs. CIL representation of JsValues. . . . 105

9.1 Hot `decompress()` method in the Compress benchmark. . . . 114
9.2 Java Grande execution times, and speedup factors. . . . 115
9.3 Spec 2008 execution times, and speedup factors. . . . 116
9.4 DaCapo execution times, and speedup factors. . . . 117
9.5 Relative speedups of C1X, MTX and MCX compared to T1X baseline. . . . 118
9.6 Relative speedups of C1X, MTX and MCX compared to Hotspot baseline. . . 119
9.7 Static Profiling Overhead . . . 120
9.8 Dynamically computed T1X code size for three instrumentation modes. . . 122
9.9 Speedup relative to code size of MTX and MCX compared to C1X baseline. 123
9.10 Code generated by the C1X compiler under three execution modes. . . . 124
9.11 Tracing Visit Metrics for Java Grande and Spec 2008. . . . 125
9.12 Tracing Visit Metrics for DaCapo. . . . 126
9.13 Trace Region Metrics . . . 127

LIST OF TABLES

Page

5.1 Message types. 67

9.2 Spec 2008 Benchmark Suite 109
9.1 Java Grande Benchmark Suite 109
9.3 DaCapo Benchmark Suite 110
9.4 Default Maxpath Options 112
9.5 Execution modes. 112
9.6 Instrumentation types. 120

ACKNOWLEDGMENTS

Six years ago I unknowingly walked into a course on advanced compiler construction. By the end of the first lecture, the professor's infectious enthusiasm had taken hold, I was hooked. In the last lecture, the professor pulled my aside and with a few brief words of encouragement convinced me to spend the next six years exploring the wondrous black art of compiler construction. Working under his supervision has been one of the most fulfilling period of my life, and for that I am forever grateful to my advisor, Professor Michael Franz.

I would also like to thank Professor Brian Demsky, Ian Harris and Tony Givargis who accepted to serve on my committee.

I would also like to thank Dr. Andreas Gal with whom I worked closely on much of the work in this dissertation and Dr. Christian Wimmer for his guidance and insightful comments. I am also very grateful to Dr. Stefan Brunthaler and Dr. Per Lersen without whom this dissertation would be unintelligible.

Furthermore, I would like to express my gratitude towards those that have made graduate school a truly wonderful experience and from whom I've learned more than I can over hope to impart on others, especially: Cindy Rubio González, Babak Salamat, Gregor Wagner, Mason Chang, Christoph Kerschbaumer, Karthik Manivannan, Eric Hennigan, Todd Jackson, Andrei Homescu.

Lastly, I would like to thank Dr. Bernd Mathiske and Doug Simon. For years, their artistry in software design has inspired me to write better code.

CURRICULUM VITAE

Michael Alexander Bebenita

EDUCATION

University of California, Irvine 2006 - 2011 (Expected)
Ph.D. in Computer Science (Systems Software) Advisor: Michael Franz

California State University, Long Beach 2000 - 2005
Bachelor of Science in Computer Science
Minor in Mathematics

RESEARCH EXPERIENCE

Graduate Research Assistant **2006–2011**
University of California, Irvine *Irvine, California*

- Lead research efforts in building Maxpath, a trace-based compiler for the new Sun Microsystems Research Java VM (Maxine)
- Developed Hotpath, a prototype trace-based JVM written entirely in Java
- Invented and explored the concept of nested trace trees to curtail the negative side effects of tail duplication in trace-based compilers
- Explored trace compilation in the context of dynamically typed languages, results eventually led to the development of the Mozilla TraceMonkey JavaScript VM and the Adobe Tamarin-Tracing ActionScript VM
- Collaborated with Sun Microsystems, Adobe, and Mozilla on trace-based compilation
- Taught various undergraduate courses on compilers and trace-based compilation
- Supervised graduate and undergraduate student research

PROFESSIONAL EXPERIENCE

Mozilla Labs Rust Project Intern
Mountain View, CA Summer 2010

- Worked on parallelism and concurrency for the Rust systems programming language
- Worked on lightweight task (coroutines) scheduling, and synchronization
- Architected a message passing system for inter task communication using lock-free queues

Microsoft Research Spur Project Intern
Redmond, WA Summer 2009

- Architected an optimizing trace-based compiler for the Spur JavaScript / CLR VM
- Worked on alias analysis, guard reduction, loop hoisting and value boxing elimination

Sun Microsystems Labs — Maxine Project Intern
Menlo Park, CA — Summer 2008

- Designed and implemented a trace-based compiler for the Maxine JVM
- Devised trace recording techniques for non-interpreted execution environments

Visa USA & Visa International — Software Developer / Consultant
Foster City, CA — 2000 - 2006

- Architected and built credit card data processing systems

Conexant — Software Developer
Newport, CA — 2000 - 2001

- Wrote tools for managing data centers and performing data migration across Windows NT domains

PUBLICATIONS

Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter; SPUR: A Trace-Based JIT Compiler for CIL; OOPSLA 2010, Reno, Nevada; October 2010

Michael Bebenita, Mason Chang, Gregor Wagner, Christian Wimmer, Andreas Gal, Michael Franz; Trace-Based Compilation in Execution Environments without Interpreters; The 8th International Conference on the Principles and Practice of Programming in Java 2010 (PPPJ 2010), Vienna, Austria; September 2010

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Michael Bebenita, Mason Chang, Michael Franz, Edwin Smith, Rick Reitmaier, and Mohammad Haghighat; Trace-based Just-in-Time Type Specialization for Dynamic Languages; Programming Language Design and Implementation (PLDI) 2009

Mason Chang, Edwin Smith, Rick Reitmaier, Andreas Gal, Michael Bebenita, Christian Wimmer, Brendan Eich, and Michael Franz; Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications; In Virtual Execution Environments (VEE); Washington, D.C, 2009

Christian Wimmer, Marcelo Cintra, Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz; Phase Detection using Trace Compilation; The 7th International Conference

on the Principles and Practice of Programming in Java 2009 (PPPJ 2009), Calgary, Alberta; August 2009

Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz; Stream-Based Dynamic Compilation for Object-Oriented Languages; The 47th International Conference on Objects, Models, Components, Patterns (TOOLS-EUROPE 2009), Zurich, Switzerland; June 2009

Michael Bebenita, Andreas Gal, Michael Franz; Implementing Fast JVM Interpreters In Java Itself; Principles and Practices of Programming Java (PPPJ 2007)

Andreas Gal, Michael Bebenita, Michael Franz; One Method At A Time Is Quite a Waste of Time; Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS2007)

Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal and Michael Franz; Efcient Just-in-Time Execution of Dynamically Typed Languages Via Code Specialization Using Precise Type Inference

ABSTRACT OF THE DISSERTATION

Trace-Based Compilation and Optimization in Meta-Circular Virtual Execution Environments

By

Michael Alexander Bebenita

Doctor of Philosophy in Computer Science

University of California, Irvine, 2011

Professor Michael Franz, Chair

Most just-in-time compilers for object-oriented languages operate at the granularity of methods. Unfortunately, even "hot" methods often contain "cold" code paths. As a consequence, just-in-time compilers waste time compiling code that will be executed only rarely, if at all. This dissertation explores an alternative approach in which only truly "hot" code paths are ever compiled. As a result, our technique compiles significantly less code and improves the performance of both statically and dynamically typed programming languages.

Chapter 1

Introduction

The popularity of virtual execution environments has risen dramatically with the advent of the Java virtual machine. Benefits such as automatic memory management, platform independence, and memory safety have helped to make the virtual machine (VM) the execution model of choice. Java, C#, JavaScript, Python, and PHP are among the most popular languages in use today [TIO11], and they all rely on virtual execution environments.

The Java programming language is used in a wide variety of contexts; from enterprise systems on high-end server hardware, to client applications on resource-constrained mobile devices, and even as a foundation for many other programming languages. The Java virtual machine must accommodate both extremes of this spectrum in a way that strikes a very careful balance between peek performance and execution latency.

For years, this has been an active area of research which has led to many advancements in dynamic compilation. Today's Java virtual machines routinely use method-based dynamic compilation with adaptive optimizations in order to boost performance and provide a low startup latency. In broad strokes, dynamic compilation systems execute applications in several phases:

- They begin executing an application using an interpreter or a simple non-optimizing compiler. This gives them an opportunity to profile the execution of the application and detect which parts are most frequently executed and would thus benefit most from compilation.
- Frequently executed regions of code, usually methods, are subsequently compiled by an optimizing compiler and linked in for future execution.
- At any moment, the dynamic compilation system can readapt and swap out previously compiled code for new code that is better suited for the current execution profile.

The general trend is to lower the granularity at which code is compiled: from whole applications at a time to single modules and methods, and even code regions within methods. Doing so, allows a dynamic compilation systems to ever so slightly adapt and specialize

compiled code to its execution profile. In this dissertation we follow this trend and take it one step further. We break down method boundaries and focus on frequently executed program paths, specializing code not only to its control flow execution profile, but also on the concrete values and types on which it operates.

My thesis is guided by a single, simple intuition:

> *program paths that were executed before are likely to be executed again*

Focusing all of our efforts on the things that are likely to happen, at the expense of those that are not, may allow us to save time and space and to do a better job compiling code. However, with over-specialization comes a loss of diversification. What are we to do if we are wrong, what if what happened before does not happen again?

This dissertation answers these questions and explores a technique called trace-based compilation where frequently executed control flow paths (traces) are detected, recorded, compiled and executed. We investigate each of these four phases in two distinct trace-based compilation prototypes that we built in support of this dissertation:

- J^2VM is a partially meta-circular Java virtual machine built to support trace-based compilation from the ground up. It is implemented as a thin layer on top of an existing Java virtual machine (hence partially meta-circular) that delegates much of the low-level work to an underlying host JVM and provides the necessary abstractions for building a trace-based compiler. Chapter 4 explores a prototype trace-based compilation system in this context and chronicles earlier work and its research influence.
- Maxpath is a complete trace-based compilation system built atop the fully meta-circular Maxine Java virtual machine. Chapter 5 examines trace-based compilation in this fully meta-circular environment where even the trace-based compiler itself is subject to trace-based recompilation and optimization. We also show how a Java virtual machine can be retrofitted to integrate a trace-based compilation system with relatively little effort. In chapter 6 we present a novel trace-driven incremental static single assignment (SSA) form construction technique especially designed in support of trace-based compilation.

In this dissertation we also asks the question: what types of client applications are especially suited for trace-based compilation? In particular, we focus on dynamically typed languages that are layered on top of statically typed languages such as Java and C#. Chapter 7 examines how trace-based compilation can be used to automatically generate type specialized code fragments for dynamically typed languages.

Chapter 2

Background

2.1 Virtual Machines

Most VMs in use today make tradeoffs between execution latency and performance (figure 2.1 on the next page). VMs designed for long-running server applications are usually concerned with peak performance and tend to ignore startup cost (latency). These VMs often use a *compiler only* approach where methods are compiled ahead of time or just before they are first executed. Since compilation time is insignificant compared to the overall execution time of a long-running server application, the system can afford to perform complex and time consuming compiler optimizations in order to improve long term execution performance.

At the other end of the spectrum, VMs designed for short-running client applications reduce latency by using an *interpreter only* design. Interpreters offer low execution latency and a small memory footprint, which makes them ideal for mobile devices and short-running applications, but suffer from poor long term execution performance.

Ideally, a VM should exhibit the performance of compiled code and the latency and memory footprint of interpreters. Over the years, VM engineers and researchers have explored ways in which both of these execution modes can be mixed to provide low latency, and even faster execution performance than compiler only approaches.

One of the main advantages of virtual execution environments is introspection. Using introspection, a VM can collect profile data about a program's execution, and use this information to adapt to its runtime behavior. The general approach is to first execute programs using an interpreter and then selectively compile frequently executed program regions for efficient long term execution.

The working assumption is that a program's execution time is spent in a very small part of the program, also known as the 90/10 law [HPG03a]: 90% of the execution time of a computer program is spent executing 10% of the code. This suggests that only the small fraction of a program which is frequently executed benefits from compiler optimizations,

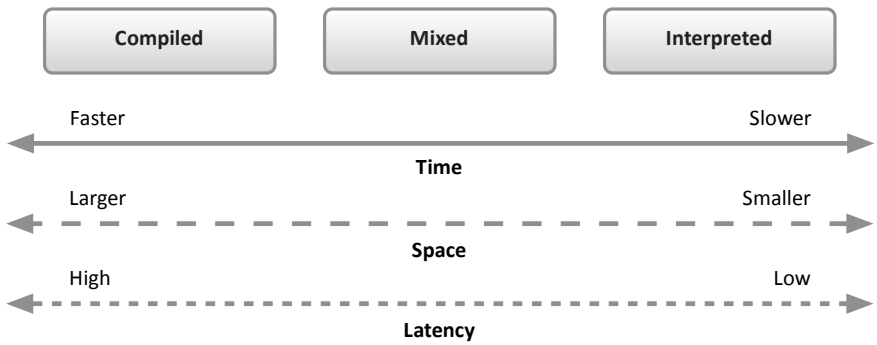


Figure 2.1: Time, space and latency tradeoffs of compilers and interpreters.

while the rest of the program is better left to the interpreter. Choosing how and when to use just-in-time compilation and when to use interpretation is a difficult engineering problem, due to several tradeoffs:

- Compiled code runs faster but also requires more space. VMs typically operate on a virtual machine instructions which implicitly encode a lot of semantic information. Lowering these instructions to machine code usually requires more space.
- An ahead-of-time compiler can devote an arbitrary amount of time to program analysis and optimization, a just-in-time compiler however has a limited compilation budget and must carefully balance code quality with compilation speed.
- Interpreted programs run slower, but require less space because they operate on a more compact virtual machine instruction format, as opposed to compiled programs which run faster, but which also require more space. If the two execution modes are mixed, then application code may need to be stored in both virtual and native instruction formats.
- Interpreters are more portable than compilers. Interpreters can be ported to another platform as long as the language the interpreter is implemented in is available on the new platform.
- Interpreters are more flexible than compilers. It is a lot easier to modify, profile, and access the state of an interpreted program than that of a program that is compiled to native code.
- Interpreters are quick to start, but slow to finish. Before a method can be executed, a just-in-time (JIT) compiler must wait for compilation to finish, an interpreter can start execution without delay.

2.1.1 Short History of Just-in-Time Compilation

The idea of dynamic compilation, or JIT compilation, which refers to the notion of translating a program to machine code after it has started execution has a rich history. The earliest

```
1 static void main(String[] args) {
2     for (int i = 0; i < 16; i++) {
3         System.out.println(i);
4     }
5 }
```

(a) A simple Java program loop.

```
1  static void main(java.lang.String[]);
2  Code:
3   0:  iconst_0
4   1:  istore_2
5   2:  iload_2
6   3:  bipush 16
7   5:  if_icmpge 21
8   8:  getstatic
9         System.out:Ljava/io/PrintStream;
10  11: iload_2
11  12: invokevirtual
12        PrintStream.println:(I)V
13  15: iinc    2, 1
14  18: goto    2
15  21: return
16 }
```

(b) Disassembled Java bytecode program.

Figure 2.2: Java Bytecode

software systems to use JIT compilation date back to the 1960s, chronicled eloquently by John Aycock [Ayc03].

The earliest mention of JIT compilation dates back to McCarthy's work on LISP [McC62] where he mentions that functions can be compiled at runtime as needed:

> The programmer may have selected S-functions compiled into machine language programs put into the core memory. Values of compiled functions are computed about 60 times as fast as they would if interpreted. Compilation is fast enough so that it is not necessary to punch compiled program for future use.

The manual for the IBM 7090 system [IBM] several years later in 1966 states that the system's assembler and loader can be used translate and load code during execution. Of particular relevance to our work is the LC^2 system by Mitchell et al. [Mit70] in 1970. Mitchell first introduced an idea similar trace-based compilation. He observed that programs can be translated to machine code by recording their execution through an interpreter, and storing the actions performed by the interpreter into a buffer, rather than executing them. Program paths that were not explored by the interpreter during translation were executed by re-invoking the interpreter, a concept familiar to many trace-based JIT systems.

The APL system by Abrams [Abr70] in 1970 used JIT compilation for optimization. APL is a dynamic language, types and attributes are usually not known until runtime, and therefore it makes sense to delay compilation until the last possible moment, when more type information is available.

Although it is a statically typed language, delayed compilation also applies to Java, but in a slightly different form. The lazy loading and linking semantics of Java require that classes are loaded and initialized at first use, not ahead of time. This is often implemented using an initialization guard that is inserted wherever a class is accessed, which checks whether the accessed class is loaded and initialized. If the class is already initialized when the method

accessing it is compiled, then there is no need to insert an initialization guard. Waiting for the JVM to reach a steady-state (when most of the application has been initialized) before compiling code eliminates much of this superfluous code.

In 1973, Darkin and Poole [DP73] describe the MITEM text editor system where they explore mixed code execution and adaptable code generation. In 1974, Hansen [Han74] explored dynamic optimizations in FORTRAN by using frequency-of-execution counters to detect and compile program "hot spots". In 1985, Smalltalk [GR83] used a JIT to compile procedures lazily, before their first execution. The Self system [CU89], several years later in 1989, described a JIT compiler where methods could be specialized to call sites. A method could be compiled several times, differently, depending on the caller's context. More recently, Java, C#, JavaScript, Python, and many other languages use JIT compilers for program optimization.

2.1.2 The Java Virtual Machine

The research presented in this dissertation is almost entirely within the context of the Java programming language; in this section we present several of its characteristics.

Java is a statically typed, object oriented programming language. Java applications are first compiled to an intermediate virtual instruction set called Java bytecode, or JVML (Java Virtual Machine Language). The `javac` source compiler is used to translate Java source files to Java bytecode and package them in `.class` files, which are subsequently executed by the Java virtual machine. In this dissertation, the term "compiler" only refers to the translation of Java bytecode to native machine code and not to the translation from Java source to bytecode.

Java bytecode provide a stack based model of computation. Each Java method has a fixed number of local variables, and an operand stack with a known stack height. At a program point p we define the program's state at that point as s_p, where s_p is the state vector $\langle p_{bci} : l_1, l_2, ..., l_n \mid t_1, t_2, ..., t_{k-1} \rangle$, and where p_{bci} is the bytecode index, n is the number of local variable slots and k is the current stack height (which is always less than or equal to the maximum stack height). Bytecode operations read their operands from the stack and push their results back on the stack. Figure 2.3 on the following page displays a short sequence of bytecode instructions that implement the Java statement `y = 2 + x`. A slightly more complicated example is shown in figure 2.2 on the previous page.

There are roughly 200 Java bytecode operations, operating on 5 primitive types: `integer`, `long`, `float`, `double` and `reference`. There are also 4 additional types: `byte`, `char`, `short`, and `boolean` which are used for field types and conversion operations. Java bytecode supports: loading and storing of local variables and fields, arithmetic, type conversions, object creation, control transfer, stack manipulation, method invocation, concurrency and exception handling.

The JVM starts execution of a Java program by loading the `.class` file containing its main method (`static main (String[] args)`). Class loading includes a bytecode verification [Ler03] phase which ensures that Java bytecode is "safe", as a way to protect against

Bytecode	Stack Operation	State After	Description
`ICONST_2`	$\mapsto 2$	$\langle x, y \mid 2 \rangle$	pushes int constant 2 onto the stack
`ILOAD_0`	$\mapsto value$	$\langle x, y \mid 2, x \rangle$	loads int local variable 0 onto the stack
`IADD`	$value_1, value_2 \mapsto result$	$\langle x, y \mid 2 + x \rangle$	adds two int values
`ISTORE_1`	$value \mapsto$	$\langle x, 2 + x \mid \rangle$	stores value into local variable 1

Figure 2.3: A sequence of bytecode operations for the Java statement `y = 2 + x`.

maliciously crafted bytecode. Once the main class is loaded and execution begins, additional classes are loaded on demand as they are referenced during execution. This is important because it allows the JVM to only load the portions of the Java Runtime Environment (JRE) libraries that are actually used. Otherwise, since the JRE is very large, loading it all at once would cripple the JVM's performance.

2.1.3 Interpreters

Early JVMs relied solely on interpretation for the execution of bytecode. This is due to the ease-of-implementation and portability that interpreters offer, considerations that were key to the designers of Java. In fact, Java bytecode were designed to be directly interpretable, and not as an abstract intermediate language. [1]

Java interpreters are at least an order of magnitude slower than code that has been compiled. This unfortunate circumstance is largely due to the instruction dispatch overhead that plagues interpreters, an overhead that has become more dramatic with the advent of pipelined processor microarchitectures. A bytecode interpreter, much like a microprocessor must repeatedly *fetch*, *decode* and *execute* instructions. This is usually done in an interpreter loop where a program counter variable is repeatedly incremented and used to fetch new bytecode to be executed. Once a bytecode instruction is fetched, depending on the bytecode, control is transferred to a subroutine that further decodes the instruction and implements its semantics. This transfer of control is called *instruction dispatch*, and is usually implemented as a native indirect branch instruction. This indirect branch is often mispredicted, and is thus likely to incur a significant number of mispredicted branches, which in turn have a severe effect on the processor's instruction pipeline. A branch misprediction causes the processor to flush its pipeline. The overhead of fetching and dispatching bytecode is very large in relation to the actual implementation of the bytecode operation, which in the case of typed Java bytecode, is usually very small.

Optimizing instruction dispatch in interpreters involves several techniques:

1. Reducing the number of instructions by increasing the semantic level of each instruction is a way to minimize the cumulative overhead of instruction dispatch. The most

[1] This is evident in the awkward treatment of `long` and `double` data types, which are 64-bit values and thus both take up two stack slots in order to satisfy the 32-bit stack slot size rule specified in the JVM specification. JVM designers originally intended for `.class` files to be directly executable, and not be subject for further pre-processing in the JVM, and thus polluted the bytecode specification with machine level constraints.

common way of doing this is to use a register based virtual instruction set architecture rather than a stack based one. Stack based bytecode implement dataflow implicitly through the operand stack. This however, increases the number of necessary bytecode instructions, in figure 2.3 on the preceding page, four bytecode instructions are needed to implement the Java statement `y = 2 + x`. Using a register based instruction format, this can be accomplished with just one instruction, `IADD y, #2, x`. The disadvantage is that the instruction can no longer be encoded with just one byte, and most likely requires at least 4 bytes, one for the instruction operation and 3 for its operands [SCEG08].

2. Reducing the number of bytecode instructions by constructing super instructions [EG03]. The idea is to replace commonly occurring sequences of bytecode instructions with fewer, but larger instructions that do the same thing. This can be done at varying levels of granularity:

   - *instruction level*: Two or three bytecode instructions are fused into one.
   - *basic block level*: Entire basic blocks [2] are replaced by compiled versions [PR98].
   - *trace level*: Long program paths that span multiple basic blocks are recorded and replaced with compiled versions.

3. Optimizing each individual *instruction dispatch*. One technique is to improve branch prediction by replicating the dispatch code across each instruction implementation. Doing so allows the processor's branch predictor to "adapt" to patterns in the interpreter's instruction stream.

   Other techniques such as threaded code [Bel73] aim to remove instruction dispatch altogether by emitting executable code. However, whether threaded code interpreters are really interpreters is debatable.

2.1.4 Just-in-Time Compilers

Even the most sophisticated interpreters still lag in performance when compared to natively compiled code. Fortunately, there is a wide variety JIT compilation techniques to choose from; ranging from simple single pass translators to heavy weight adaptive optimizing compilers (figure 2.4 on page 10).

Call Threaded Code This is the most primitive way of building a just-in-time compiler and resembles an interpreter. The major distinction is that, instead of using an interpreter dispatch loop, code is generated that invokes each of the instruction implementation routines in program order. If we assume the straight sequence of virtual instructions $v_c, v_b, v_a, \ldots, v_b, v_a$ implemented by the routines $r_c, r_b, r_a, \ldots, r_b, r_a$ respectively, a thread code JIT generates the sequence of native `call` instructions `call` r_c, `call` r_b, `call` r_a, ..., `call` r_b, `call` r_a. Execution of the program just becomes a matter of starting execution at the beginning of this generated code sequence. This is faster than an interpreter because there is no instruction dispatch, and no instruction decoding.

[2] A basic block is a block of code that has only one entry point and only one exit point.

Inline Threaded Code As an optimization to call threaded code, instead of generating calls to implementation subroutines, the subroutines themselves are inlined into the generated code. This is also sometimes called macro expansion, or template-based compilation because the JIT simply substitutes a virtual instruction with the native code of the subroutine that implements it. In order to be able to do this, the subroutine must follow a special calling convention so that it does not create and tear down its own activation record. Usually, these subroutines are assembled by hand, but may also be generated from source code. The latter requires that the JIT has intimate control over how the subroutine is compiled by the host compiler, which is not always possible. Our research on trace-based compilation is within the context of this type of baseline JIT compiler, as is presented in depth in chapter 5 on page 44.

Optimizing Compilers Optimizing compilers such as the Java HotSpot Client Compiler [KWM+08] are sophisticated pieces of software, orders of magnitude more complex than the previously mentioned JIT compilation techniques. Optimizing compilers transform programs from their virtual instruction set representation to a more malleable *high-level intermediate representation* (HIR) which is suitable for compiler optimizations. The HIR is subsequently processed by various optimization phases before lowered to a machine dependent *low-level intermediate representation* (LIR) and handed off to the *register allocator* whose responsibility is to map virtual registers/variables onto machine registers. Finally, the LIR is transformed into native code and executed.

The state of the art in JIT compilation for the Java platform uses this architecture. What sets just-in-time compilers apart from their ahead-of-time counterparts, is their close interaction with the rest of the runtime system. By observing the execution of a client application, the Java virtual machine can adapt to its runtime behavior. This allows the just-in-time compiler to perform many feedback directed optimizations which would not otherwise be possible ahead of time. Feedback directed optimizations [AFG+05] such as deep inlining and the de-virtualization of virtual methods are crucial to Java performance.

Deoptimization De-virtualization is the process of converting polymorphic call sites into monomorphic call sites, thereby reducing virtual calls into direct calls. Normally, a static compiler cannot predict a priori whether a polymorphic call site is monomorphic at runtime. For example, a Java method that is not marked as final [3] and whose enclosing class is also not marked as final may be overridden at any time in the future during an application's execution. This possibility prevents calls to this method from being de-virtualized. A JIT compiler, however, can do something very clever. Since it has access to runtime information, it can perform a class hierarchy analysis and determine whether the call site is "currently" monomorphic and de-virtualize it. In the future, if this assumption is invalidated by new classes being loaded, the compiled code is deoptimized and invalidated (or thrown away). [4]

Deoptimization is the process of interrupting a program execution while in optimized code and resuming it in the interpreter. Dynamic deoptimization was used in Self [HCU92] to allow for source-level debugging of globally optimized code. Once code is optimized it loses

[3] The `final` keyword in Java indicates that a method is not overridable or that class cannot be subclassed.
[4] This is similar to polymorphic inline caches (PICs), but does not require the use of runtime type guards.

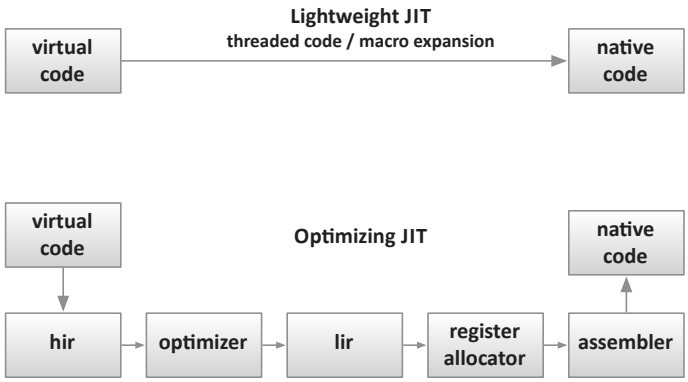


Figure 2.4: Lightweight vs. Optimizing JIT Compilers

its relationship with the originating source code. Inlining, register allocation, constant propagation and other compiler optimizations produce code that is no longer in correspondence with the program's source code. A possible solution to this problem is to always execute the application without any optimizations whenever debugging. This is not always acceptable because the debugged application does not exhibit what the SELF authors refer to as the *expected behavior*, the behavior it has under normal execution. In SELF, deoptimization was used to interrupt the execution of optimized code whenever a debug breakpoint was hit, and resumed in interpreted code where normal debugging could be performed. Deoptimization requires that extra recovery information is stored in optimized code that allows the system to reconstruct the activation records for the interpreter [Höl94].

Our own research on trace-based compilation is related to deoptimization. The crucial distinction between the deoptimization approaches used in modern VMs and our trace-based compilation technique is that we perform deoptimization on the critical path and not as an uncommon occurrence. We use it at a fine grained level, to speculate on actual control flow paths and runtime values rather than on coarse grained events like breakpoints and code invalidation requests.

2.2 Trace Optimizations

A *trace* is a sequence of instructions spanning one or more basic blocks. Traces have been used in program optimizations for nearly 30 years. If chosen wisely, traces capture frequently executed program paths and ignore infrequently executed program regions. Focusing compiler optimizations on the critical execution path has always been a wise choice, as is evident in the use of traces at every level of computer architecture.

2.2.1 Trace Cache

At the microarchitecture level, traces have been used to speed up the fetching and decoding of instructions. The Pentium® 4, NetBurst™ microarchitecture includes a new form

of instruction cache called the *Execution Trace Cache* [SGC01]. The trace cache is the Level 1 instruction cache for the Pentium 4 execution pipeline. Because of their format, IA-32 instructions are difficult to decode. Before execution, the processor converts IA-32 instructions into several simple μops and caches these simpler operations in the trace cache. The trace cache holds up to 12K μops. A trace cache miss results in a new IA-32 instruction being decoded from the L2 cache.

Each trace cache line holds up to 6 μops. The trace cache assembles sequences of μops called traces that run down the predicted path of the IA-32 program execution and packs them into several trace cache lines. This allows for the target of a branch to appear in the same cache line as the branch itself, even if they are not close to each other in the original IA-32 program.

The main advantage of the trace cache is that it can deliver μops up to, and after a branch instruction, including the branch target as well. With a conventional cache, the processor would have to wait for the branch predictor before fetching the branch target cache line. The trace cache has its own branch predictor, but the predictor only needs to deal with the subset of the program that is in the trace cache, and not with the entire program. [5]

2.2.2 Trace Scheduling

Very Long Instruction Word (VLIW) architectures are underutilized by most compilers. Their large number of parallel and pipelined functional units are not sufficiently exploited by a single thread of control. To resolve this problem, compilers use instruction scheduling to rearrange program instructions in a way that makes better use of processor resources and yet preserves program behavior. The general technique is to reduce data hazards such as: *Read-after-Write*, *Write-after-Read* and *Write-after-Write* all of which can stall the processor's execution pipeline, as well as to reduce hazards resulting from multiple instructions competing for the same hardware resources. Scheduling is usually performed independently on each basic block by constructing an instruction dependency graph where each edge is labeled with a latency cost, and then computing a topological ordering that minimizes the total latency cost. The problem with this approach is that basic blocks are usually quite small and contain very little instruction level parallelism. [6] Performing global instruction scheduling on multiple basic blocks is difficult because control flow dependencies are introduced and have to be taken into account.

Trace scheduling [Ell84] is a technique that combines several frequently executed basic blocks together into a trace and performs instruction scheduling across this larger block of code. The drawback is that instructions that are moved across the original basic block boundaries need compensation code in off-trace basic blocks to undo any of their side effects in case program execution exits the trace.

Similar to our trace-compilation work, trace scheduling sacrifices the performance of off-

[5] The Pentium 4 architecture was eventually discontinued because of its large execution pipeline, an issue unrelated to the trace cache.

[6] Research [HPG03b] has show that there is generally too little instruction level paralleleism to justify VLIW architectures.

trace basic blocks for the performance of on-trace blocks. Unlike our system, trace scheduling can be applied to any ahead-of-time compiler infrastructure, and does not depend on runtime feedback for the selection of traces; they are usually selected through static program analysis at compile time.

2.2.3 Dynamic Binary Translators

Binary translation is the emulation of an instruction set using another through the translation of binary code. Dynamo [BDB00] was the first dynamic binary translator to make use of trace-based optimizations. It improved the performance of statically compiled native binaries for the HP PA-8000 processor by first interpreting (or emulating) native instructions, recording frequently executed program traces, and then recompiling these into a more efficient form for the same target architecture. The surprising result was that Dynamo was able to offset the overhead of its own execution (1.5% as reported by the authors) and improve the performance of applications by 9% on average. It performed best on non-optimized binaries, and performed well on optimized binaries but lost its benefits once binaries were compiled with static profile guided optimizations.

Using an interpreter, Dynamo profiled the execution of backwards branches and once a certain "hot" threshold was met, it would start recording traces in a hot trace buffer until an "end-of-trace" condition was met. The recorded trace was subsequently optimized and compiled into a single-entry and multiple-exit sequence of code called a "fragment" and finally stored in a "fragment cache". Backward branches, or branches to already existing fragments in the fragment cache were considered "end-of-trace" conditions. This allowed fragments to be linked together as they accumulated in the cache, but also required that fragment entry and exit points retain their original machine register mappings so that the application's machine state would be compatible when control was transferred from a fragment's exit-point to another fragment's entry-point.

If a fragment's exit-point was not linked to another fragment then a "transfer" would occur. Control was transferred to the interpreter, which would take a snapshot of the current machine state, use it for interpretation, and restore it back once control was transferred back to a new trace fragment.

2.2.4 Summary

The main conceptual distinction between the previous approaches and our work is that we operate on a much higher semantic level. These approaches operate on low-level native instructions with intricate semantics and side effects, and must always stay true to the machine state, which considerably limits their use. The length of these traces is usually very small, and few optimizations are possible. Our work raises the level of abstraction considerably. We operate on high-level virtual instructions with clear but much higher level semantics, such as those of the Java virtual machine.

Chapter 3

Evolution of Trace Regions

Representing Partial Programs using Traces

Using runtime profiling information, a just-in-time compiler can focus its optimization budget on regions of code that contribute to the overall runtime of an application and ignore those that do not. Because of their importance to performance, many just-in-time compilers focus on loops, and now even optimize frequently executed paths within loops [GES+09, BBF+10, BCW+10], giving rise to a recent research thrust on *trace-based compilation*.

Existing approaches to loop-oriented dynamic compilation using traces often provide only limited control flow coverage. Furthermore, these existing approaches merge alternative paths through a loop at the loop header only, i.e., shared code paths downstream from a control flow split are replicated across traces. Recursive tail duplication of loop paths can result in explosive code growth. In this chapter, we introduce a more flexible region-based trace compilation approach that models control flow join points inside of loop bodies and thereby overcomes existing limitations of trace-based compilers.

To control code explosion, trace-based compilers often limit the length of traces to just a few basic blocks and link them together. The major disadvantage to this approach is that very few optimizations can be performed across these trace link points. Instead, we introduce a more flexible data structure, namely the *trace region*, that accumulates many linear traces in a more general control flow graph (CFG). As new traces are added to the trace region, they are joined to previously recorded traces, effectively curtailing tail duplication. Unlike previous approaches using trace trees [Gal06], trace regions can include join nodes, making trace-based compilation viable for large-scale applications.

3.1 A Motivating Example

Trivial operations such as appending a number to a *List* can be surprisingly complicated in languages with rich runtime libraries such as Java. For example, the code segment in

figure 3.1a adds `1000` numbers to a list that is defined as an interface (`List<Integer>`) but has an unknown concrete type. The key to improving performance in this case is inlining and speculative execution. Conventional method-based compilers try to inline the interface call to `List.add()` as well as the implicit static call to `Integer.valueOf()` [1]. Inlining the interface call requires the insertion of a guard to ensure that control is always dispatched to the speculated interface target (figure 3.1b, line 6). If the speculative guard fails, execution must fall back on another version of the program that has not attempted to inline the interface call, a process known as dynamic *deoptimization* [HCU92] (figure 3.1b, line 7, `DEOPTIMIZATION`). Method inlining is important because it can create more optimization potential. Concerns about code explosion unfortunately limits its use to relatively small methods, and most method-based compilers may not inline `ArrayList.add()` due to its large size. `ArrayList`s are backed by arrays and may need to be resized occasionally to accommodate new elements. The method `ArrayList.add()` contains code that performs this operation, which contributes to its large size, causing it not to be inlined.

Because it only focuses on small method fragments, our trace-based compiler can perform more aggressive inlining than method-based compilers can. In the tracing example shown in figure 3.1c, much of the inlined method is stripped away and only the frequently executed path is considered for compilation. If a rarely executed path is taken at runtime, the optimized trace is exited using a *transfer* (e.g. figure 3.1c, line 5, `TRANSFER`). A transfer is similar to a deoptimization, i.e., a reconstruction of the unoptimized execution state, but it occurs much more frequently, and therefore, needs to be made much more efficient. Transfers are used frequently to speculate on control flow, deoptimizations on the other hand are normally used in exceptional cases, for debugging or when compilation assumptions are invalidated by class loading.

In summary, trace-based compilation provides the following advantages for the example in figure 3.1 on the next page:

- Compilation time is reduced because only a small subset of the CFG is compiled.
- Due to partial method inlining, we can afford to inline more aggressively, exposing more potential optimizations.
- Control flow guards offer a great deal of information to the optimizing compiler. Basic blocks dominated by control flow guards are predicated on guard conditions. For example, the expression: `list.elementData[elementCount++] = value;` at figure 3.1c, line 11, is predicated on the condition `elementCount + 1 <= oldCapacity` (line 8). From this, it is easy to infer that `elementCount < list.elementData.length`. This can be used to eliminate the array bounds check. In a method-based compiler this optimization would not be possible, since the body of the `if` statement would also be compiled and control flow at line 11 may be reached through this alternative path.

[1] In Java, casting primitive `int` types to boxed `Integer` types involves a call to the `Integer.valueOf()` function which caches the most frequently boxed values.

```
/* (a) Original Program */
 0 List<Integer> list = ...;
 1 for (int i = 0; i < 1000; i++) {
 2     list.add(i);
 3 }

/* (b) Speculative Method Inlining (2 Levels) */
 0 List<Integer> list = ...;
 1 for (int i = 0; i < 1000; i++) {
 2    // Integer Conversion
 3    Integer value = i >= -128 && i <= Integer.IntegerCache.high
 4        ? IntegerCache.cache[i + 128] : new Integer(i);
 5    // Type Guard
 6    if (list.getClass() != ArrayList.class) {
 7          /* DEOPTIMIZATION */
 8    }
 9    // Inlined Method
10    int oldCapacity = list.elementData.length;
11    if (elementCount + 1 > oldCapacity) {
12        Object[] oldData = list.elementData;
13        int newCapacity = (list.capacityIncrement > 0) ?
14            (oldCapacity + list.capacityIncrement) :
15            (oldCapacity * 2);
16        if (newCapacity < elementCount + 1) {
17            newCapacity = elementCount + 1;
18        }
19        list.elementData =
20            Arrays.copyOf(list.elementData, newCapacity);
21    }
22    list.elementData[elementCount++] = value;
23 }

/* (c) Tracing (After 500 Loop Iterations) */
 0 List<Integer> list = ...;
 1 for (int i = 0; i < 1000; i++) {
 2    Integer value = i >= -128 && i <= Integer.IntegerCache.high
 3        ? IntegerCache.cache[i + 128] : new Integer(i);
 4    if (list.getClass() != ArrayList.class) {
 5          /* TRANSFER */
 6    }
 7    int oldCapacity = list.elementData.length;
 8    if (elementCount + 1 > oldCapacity) {
 9        /* TRANSFER */
10    }
11    list.elementData[elementCount++] = value;
12 }
```

Figure 3.1: Speculative method inlining vs. trace-based compilation.

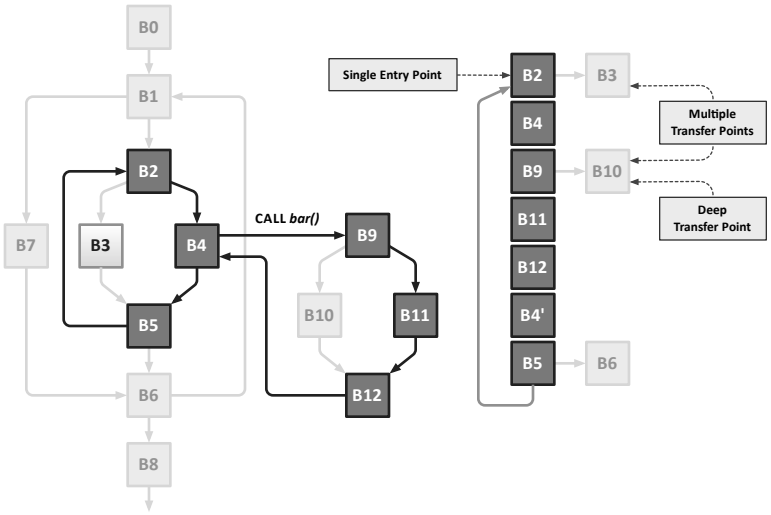


Figure 3.2: A recorded program execution trace (right) through the CFG shown on the left.

3.2 Representing Partial Programs using Traces

> *A **trace** is a sequence of successive instructions or basic blocks observed during the execution of a program.*

Traces can begin and end at any two program points; however we usually only consider those that begin and end at loop headers. Control flow to off-trace basic blocks cause trace exits, which are handled by *transfers*. If a transfer is reached during the execution of a trace, control must fall back on another execution mode (usually an interpreter or a baseline compiler), at exactly the same place where the trace left off. This transfer is accomplished with the help of a special `transfer` instruction that carries with it enough program state to reconstruct the environment of another execution mode.

Figure 3.2 shows the control flow graphs of a method with two nested loops and a call to a method `bar()`. The most frequently executed path is {B2, B4, B9, B11, B12, B4', B5}, starting at the loop header of the inner loop B2 and partially inlining method `bar()`. Block B4' represents the code region following the call instruction in the original block B4, shown here for clarity as it would normally be appended to block B12. Blocks B3, B10, B6 are transfer points to off-trace basic blocks.

In essence, trace-based compilation is a speculative optimization. If the program's execution stays on trace, sufficiently long enough to gain from trace optimizations and compilation, then it pays off in terms of performance. Otherwise, due to the overhead of detecting, recording, and compiling traces, performance can actually worsen.

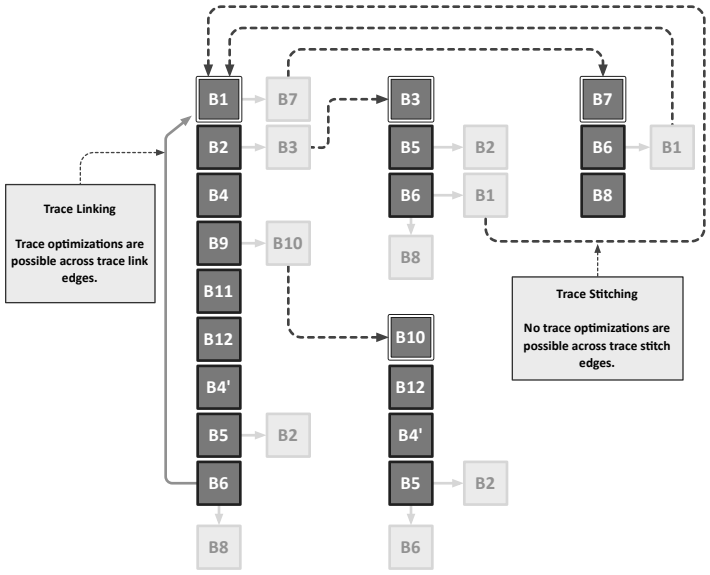


Figure 3.3: The CFG is completely covered by four overlapping traces that are recorded and compiled independently of one other. The traces are anchored at basic blocks B1, B3, B7 and B10. The trace starting with B1 loops back to its anchor block, while all others terminate elsewhere.

Trace Linking

In order to capture all frequently executed program paths, multiple traces may be necessary. However, transfering control between traces that are individually compiled requires *trace linking* and a calling convention. The CFG in figure 3.3 is completely covered by four overlapping traces which are recorded and compiled independently of one other. Control can only enter a trace through its entry point, and exit through one of its exit points. If one trace's exit point matches another trace's entry point, the two traces can be linked together. Otherwise, execution must wait until another trace's entry point is reached before entering the trace. Trace transitions pose several problems:

1. Optimizations cannot be performed across trace transitions, severely limiting their usefulness.

2. Trace transitions and transfers are suboptimal because of incompatibilities between the frame (activation record) layouts of methods and traces. Keeping the trace frame layout synchronized with the method frame would be far too expensive and would inhibit trace optimizations. Keeping the two separated allows us produce better code. It allows us to perform register allocation and avoid the creation and teardown of method frames in inlined contexts within compiled traces. The drawback is that additional adapter code is needed to copy live values in and out of traces.

3. Transitioning between traces is inefficient because live values need to be copied twice: first, from the trace frame into the method frame, and then again into frame of the next trace. The situation is further complicated if the transition occurs in an inlined

context, as is the case between basic block B9 and B10 in figure 3.3 on the preceding page. In this case, an additional method frame for the `bar()` method would have to be synthesized on the method frame stack.

However, this approach is advantageous for it allows the system to compile short traces that *collectively* cover most of the frequently executed control flow. The drawback is that traces have to be kept short, usually the size of a basic block, so the recorded paths can be linked together to resemble the original control flow graph. This renders many optimizations as well as register allocation ineffective, and the approach is often relegated to trace-based systems that would otherwise not benefit from their use. For instance, the Dynamo system [BDB00] operates at a low semantic level; it reoptimizes already optimized machine code that is not suited for aggressive global optimizations and register allocation. Our work, on the other hand, focuses on unoptimized [2] Java bytecode which has a higher semantic level. In this context, aggressive global optimizations are essential.

3.2.1 Trace Trees

It is often preferable to collect and compile several traces all at once. Doing so eliminates the overhead of trace transitions altogether, and also produces better code because optimizations are performed on several traces together. One way to collect and compile traces is to use a tree of traces.

> *A **trace tree** is a collection of cyclic traces that are all rooted at one **anchor** point, and that cover multiple program paths.*

Each trace in a trace tree starts and ends at the same program location, namely the trace tree *anchor*, or loop header. Figure 3.4 on the next page shows the growth of a trace tree anchored at basic block B2 (the inner loop header). The left-most trace is the first trace to be added to the trace tree (the same as in figure 3.2 on page 16). The second trace extends the tree at basic block B10 and duplicates blocks {B12, B4', B5} (*tail duplication*). Two additional traces are attached to cover more and more of the original control flow. The primary advantage of trace trees is that they do not contain control flow join blocks (with the exception of the first basic block). The lack of control flow join nodes simplifies many compiler optimizations, making trace trees attractive to just-in-time compilers that have a limited compilation budget.

Trace trees were first introduced in the Hotpath VM [GPF06], and later used in TraceMonkey [GES+09] and SPUR [BBF+10]. These projects focus on small kernel-based, numerically intensive benchmarks, and have not attempted to run larger-scale benchmarks.

Unfortunately, in large programs, excessive tail duplication can lead to code explosion, making trace trees unviable. Consider the following pathological example in figure 3.5a

[2] In order to aid debugging, Java bytecode produced by `javac` is in direct correspondence with Java source code and is thus unoptimized.

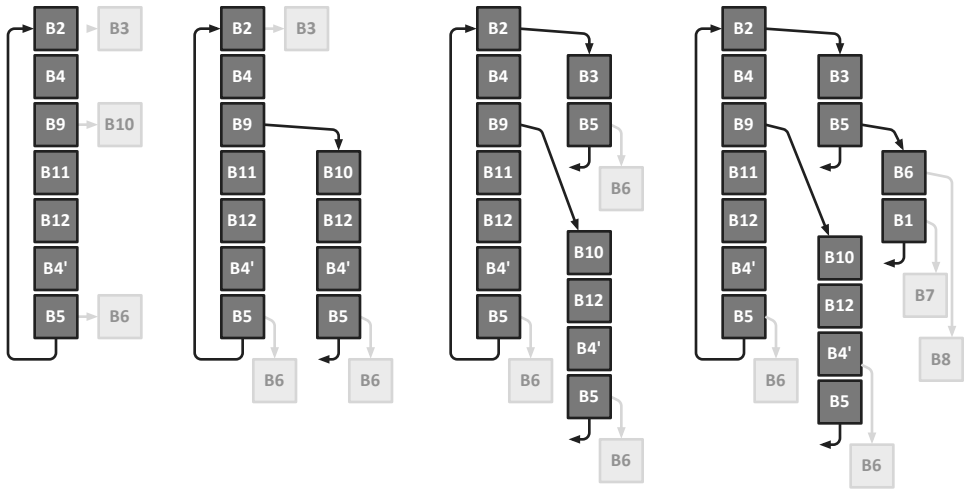


Figure 3.4: Growth of a trace tree, with excessive tail duplication.

```
for (int i = 0; i < 128; i++) {
    if ((i &  1) != 0) { ... }
    if ((i &  2) != 0) { ... }
    if ((i &  4) != 0) { ... }
    if ((i &  8) != 0) { ... }
    if ((i & 16) != 0) { ... }
    if ((i & 32) != 0) { ... }
    if ((i & 64) != 0) { ... }
}
```

(a)

```
for (int i = 1; i <= n; i++) {
    a.virtualMethod();
    switch (i) {
        case 1: ... break;
        case 2: ... break;
        ...
        case n: ... break;
    }
}
```

(b)

```
for (int i = 0; i < 1000; i++) {
    for (int j = 0; j < 10; j++) {
        ...
    }
    for (int j = 0; j < 10; j++) {
        ...
    }
}
```

(c)

Figure 3.5: Pathological cases for trace tree coverage.

where all 128 possible control flow paths are taken, causing a trace tree to grow to have 128 branches.

In addition, consider the example in figure 3.5b. Because of the `switch` statement, at least n possible branches may appear in a trace tree. If the tracer is configured to inline virtual methods, and the virtual method call has m possible targets, then the trace tree could grow to have $m \times n$ traces because of the tail duplication of the join block after the call. If the body of the inlined virtual method is in no way related to any of the individual case statements, then this exponential trace fan-out is unprofitable and merely expends the compilation budget. Gains from tail duplication are predicated on the compiler being able to perform optimizations across multiple basic blocks. If no optimization opportunities exist, there is no reason to perform tail duplication at all.

3.2.2 Nested Trace Trees

Nested trace trees are used to capture larger control flow structures that cannot be captured with trace trees alone and are a contribution of this dissertation.

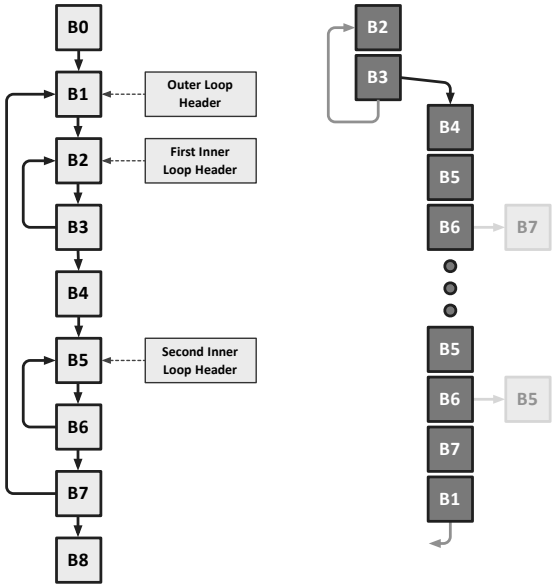


Figure 3.6: Hot loop pair (left) and a possible trace tree (right) for the first inner loop that completely unrolls the execution of the second inner loop.

> ***Nested trace trees*** *form a hierarchy of trace trees that capture nested control flow.*

Consider the example in figure 3.5c, where two equally hot inner loops are nested within one outer loop. The control flow graph for this example is shown in figure 3.6 (left). The problem in this case is that although there are three possible trace trees that can be constructed (anchored at B1, B2, B5), each of them unrolls one or both of the inner loops. If a trace tree is recorded for the first inner loop starting at B2, then additional traces attached to this tree will completely unroll the second inner loop before reaching the first inner loop header. Figure 3.6 (right) shows a trace tree in such a situation. Similarly, if a trace tree is recorded for the second inner loop starting at B5, then any additional traces will completely unroll the first inner loop. A third possibility is that a trace tree is recorded for the outer loop starting at B1, but any trace in this tree will have to completely unroll both of the inner loops. Since loop unrolling is only useful to a certain degree before it starts consuming too much of the compilation budget and code cache, it needs to limited. In short, a trace tree that completely captures this type of control flow cannot be constructed. Instead, what happens is that two trace trees with one trace each are constructed for each of the inner loops, and then invoked repeatedly every outer loop iteration. If the workload of the inner loops is small, then execution time is dominated by the overhead of transferring into and transfer out of trace trees.

In this dissertation we introduce nested trees as a way to eliminate much of this overhead and capture more control flow. When constructing a trace tree for the outer loop, instead of unrolling the inner loops, we *nest* their respective trace trees, which are likely to have already been constructed due to their higher execution frequency. Nesting is similar to procedure invocation with the exception that the *nested* tree may have many return points,

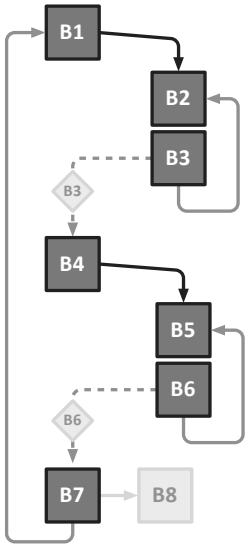


Figure 3.7: Nested trace trees.

and not just one as is the case for procedures. [3] The tree nesting process involves several steps:

1. During trace recording, if we reach an already compiled tree, we execute it and record the fact that this tree is nested along the currently recorded path.
2. After the execution of the nested tree, we resume recording in the outer *nesting* tree but at the exit point of the nested tree. Since the nested (inner) tree may have many such exit points, the nesting (outer) tree needs a control flow guard to ensure that the same exit point is always taken at this nesting location.

Figure 3.7 depicts two trees nested within a trace tree anchored at B1. The outer trace invokes each of the nested trees and guards (diamond symbol) on their exit points.

Nested trace tree provide several advantages:

- They capture significantly more complicated control flow and therefore create more optimization potential.
- As will be discussed in future chapters, trace trees can be expressed and optimized using SSA (static single assignment) form. Trace tree nesting preserves SSA form and adds little overhead to the compilation pipeline.
- Each trace tree can be compiled individually, or can be specialized based on nesting location which can happen when the nested tree is in an inlined method or along two tail duplicated program paths and is nested in different contexts.

[3] Although procedures may have many return statements, control flow is in fact joined into only one exit block thus forming only one exit point. Trace trees however, may have many control flow exit points.

- Trace tree optimizations can be propagated up and down the nesting hierarchy, allowing for trace trees to be compiled separately and yet share optimization results.
- They can improve register allocation because the hottest program regions are captured in nested leaf trees and are allocated separately.

Nested trace tree also have several drawbacks:

- Their complex structure makes them hard to reason about and difficult to implement.
- Because optimizations can cross nesting boundaries, transfer of control between trace trees in a nesting hierarchy occurs through an optimized calling convention. This is unlike trace linking, which always requires that a canonical form of the programs state is created between trace transitions. This means that trace trees can either be compiled to be callable from other trees, or callable from the interpreter or baseline compiler. Optionally they may provide two entry points depending on the calling context but this complicates the transfer process. If called from the interpreter, the transfer process must restore the interpreter's state. If called from another tree, the transfer process must make all live-out variables available to the outer nesting tree.
- If a transfer occurs in a deeply nested tree at an exit point that is not guarded by an outer nesting tree, then a chained sequence of transfers needs to occur which is difficult to implement.

Despite their implementation drawbacks, nested trace trees proven useful and have made their way into production software. The Mozilla Firefox browser, which is used by over 500 Million daily users, uses nested trace trees within its trace-based JIT compiler. Our work on nested trace trees was published at the Conference on Programming Language Design and Implementation in 2009 [GES+09].

3.2.3 Trace Regions

In this dissertation we also introduce a more flexible data structure, namely the *trace region*.

> *A **trace region** is an union of traces that is constructed dynamically, in lock step with program execution.*

A trace region is similar to a trace tree in that all traces are rooted at one program point, the trace region *anchor*. However, unlike trace trees, trace regions can include join blocks, making trace-based compilation viable for larger-scale applications. For instance, the use of trace regions would require at most 14, and $m + n$ paths for the examples in figure 3.5a and figure 3.5b, respectively.

Figure 3.8 on the following page shows a trace region following the same growth path as the trace tree shown in figure 3.4 on page 19. As new traces are added to the trace region, join blocks are inserted thus curtailing tail duplication. Trace region construction and optimization is covered in detail in chapter 6 on page 75.

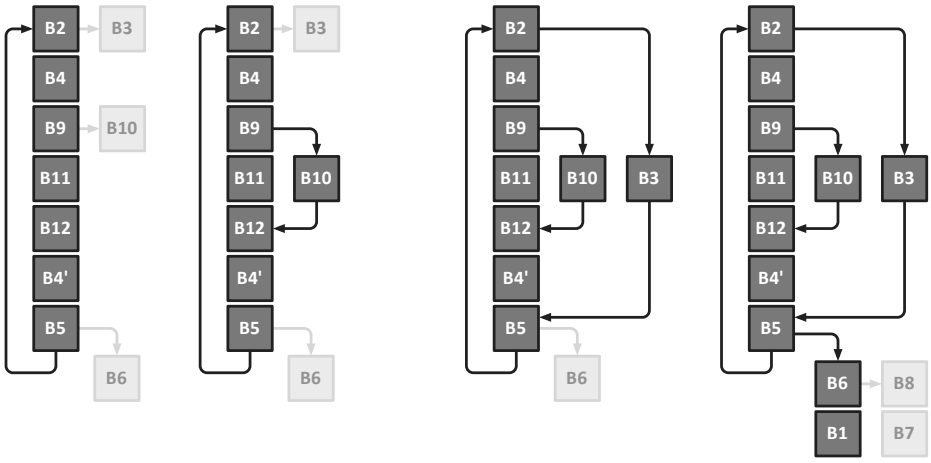


Figure 3.8: Growth of a trace region following the same growth path as the trace tree shown in figure 3.4 on page 19.

Chapter 4

J^{2}VM

Trace-Based Compilation in Partially Meta-Circular Virtual Execution Environments with Interpreters

Building a trace-based compilation system requires a significant amount of infrastructure which is not directly available in any production or research JVM today. Since this new compilation model does not fit into any of the existing VMs, we built a new prototype JVM, J^{2}VM, to support trace-based compilation from the ground up. In order to take advantage of the significant investment in current JVM technology we built the J^{2}VM system as a thin layer on top of an existing Java virtual machine.

The J^{2}VM system includes a Java interpreter written in Java, that delegates much of the low-level work to an underlying host VM, and provides the necessary abstractions for building a trace compiler. In this partially meta-circular context, we explore trace-based compilation and present a system that aims to improve the performance of Java applications, while the system itself executes on top of an host JVM as a client application.

4.1 J^{2}VM

Most Java virtual machines (JVMs) are themselves written in unsafe languages, making it unduly difficult to build trustworthy and safe JVM platforms. While some progress has been made on removing *compilers* from the trusted computing base (using certifying compilation), JVM *interpreters* continue to be built almost exclusively in C/C++. We have implemented an alternative approach, in which the JVM interpreter itself is built in Java, and runs atop a host JVM execution environment. Despite benefiting from the additional safety guarantees of the JVM runtime system, the execution overhead of our nested Java interpreter is quite acceptable in practice. Our results suggest that implementors should concentrate their efforts on optimizing just-in-time compilers rather than on interpreters. If a mixed-mode VM environment is desired, a generic JVM interpreter can subsequently be created using Java itself.

In addition to the benefit of portability, the JVM bytecode is also verifiably type-safe, allowing code consumers to check before execution that a program is well behaved with respect to certain memory and type safety properties. As a result, Java programs are generally safer and less prone to code injection vulnerabilities than programs written in legacy languages such as C.

The use of Java (and similar languages such as C# [Mic02]) has spread to almost all computing domains, from desktop applications to high-performance computing [KCSL00], and includes even Java-based operating systems [GFWK02] and operating system components [YW06]. However, when it comes to building the underlying JVM interpreters, the type-unsafe C [KR88] and C++ [Str93] languages are still the far most common implementation languages.

We present J^{2}VM, a pure Java implementation of a JVM bytecode interpreter. Implementing the JVM interpreter in Java itself has a number of advantages over the traditional C or C++ based implementation approach:

C is a weakly typed language that can be understood as a thin layer above the machine language. While C abstracts the details of the specific machine instructions, it maintains the low-level semantics of the machine with regards to the memory architecture. While this flexibility allows for the fine-tuning of memory access patterns and cache behavior, it also makes C code vulnerable to buffer overflows and memory errors. If such errors exist within the implementation of a JVM bytecode interpreter, they can be exploited by malicious Java applets to escape out of the "sandbox" of the virtual machine. Considering the size of commercial JVM implementations (e.g., Sun's Hotspot VM consists of roughly half a million lines of C++ and assembly language code), it is likely that errors will remain undetected for substantial amount of time.

By implementing the interpreter itself in type-safe Java, we can give much stronger guarantees regarding its type safety and memory correctness, increasing the overall robustness of our JVM platform. It also makes it much easier to evolve the JVM. In the case of an interpreter that is written in an unsafe language, every change to the code base necessitates a re-certification of the complete code. In the context of a high-assurance deployment environment, this can be a very costly endeavor.

Traditional C-based interpreters are pre-compiled to executable platform-specific machine code. This significantly limits the amount of runtime optimization and customization that can be done on the interpreter code. For example, the JVM Tools Interface (JVM TI) permits developers to debug and single step the execution of JVM bytecode. To enable this functionality, every time the JVM interpreter executes a bytecode instruction it has to check whether a debugger is attached and whether a breakpoint has been reached. [1] At least one check and one conditional branch is incurred for *every* instruction that is executed, even if no debugger is attached—which is almost always the case in a production environment.

Similarly, debugging code in the interpreter itself can get in the way of production performance. The Hotspot VM, for example, allows execution in a more verbose mode that

[1] Several separate interpreter dispatch routines can be used to circumvent this overhead, but that may not always be preferred.

makes it easier to diagnose VM errors more easily. However, since the VM is pre-compiled and such diagnostic code cannot be disabled without a runtime penalty, the VM can be built in several different configurations with various debugging and optimization features enabled or disabled. To reduce the runtime overhead in configurations that support such optional features, Hotspot then self-modifies its own machine code dynamically in memory to remove or insert conditional counters or debugger invocations in the already compiled interpreter code.

A Java-based interpreter is much more flexible in this regard, because its shipped in bytecode form. Once running, it is subject to just-in-time compilation just like any other Java code in the system. Using speculative dead code elimination, optimizing JIT compilers can remove any overhead associated with debugging code if the VM is started with its debugging capabilities disabled. Once debugging is enabled, the VM can recompile the interpreter and reintroduce debugging code. No machine-specific native code manipulation is necessary, and the entire interpreter and all optional parts can be implemented in pure high-level Java.

J^{2}VM's enhanced safety, configurability, and extensibility does not come entirely for free. Some of the added type-safety and memory-safety checks that are implicitly provided by Java have to be performed at runtime. This causes our interpreter to be approximately 4 to 6 times slower on computationally intensive code than a comparable C-based implementation. However, in practice, the majority of performance-critical code is compiled directly into native code and hence not run by our slower interpreter. As a consequence, for typical workloads of today's JVMs, the overall performance degradation that results from using a JVM interpreter written in Java is not usually noticeable.

4.1.1 The J^{2}VM Interpreter

J^{2}VM is a JVM interpreter written in pure Java that runs on top of a traditional Java virtual machine such as Sun's Hotspot JVM or IBM's J9 VM, which we call the *host VM*. J^{2}VM executes application programs by interpreting JVML bytecode. The overall architecture of a system incorporating J^{2}VM is shown in figure 4.1 on the next page.

For our investigation, we used Sun's Hotspot JVM for 64-bit x86 (AMD64) as the host VM. While the Hotspot VM contains a Java bytecode interpreter of its own, this is not necessary for our system. The intended application of our pure Java-based JVM interpreter is to be deployed in a system that only features a Java to native code compiler such as IBM's Jikes Research VM. Our Java-based JVM interpreter can then run on top of such a compilation-only system to provide low-latency execution for non performance-critical code.

By using a mixed-mode VM such as Hotspot for our experiments, we slightly underestimate the performance of our own interpreter since initially our interpreter is run by the Hotspot interpreter (double interpretation), which results in a severe performance degradation. However, the interpreter loop of J^{2}VM is quickly detected to be "hot" and is subsequently compiled into directly executable machine code by the Hotspot JIT compiler. For sufficiently long running programs, the performance that we measure when using Hotspot as our host VM should be reasonably close to directly compiling VMs (such as Jikes RVM).

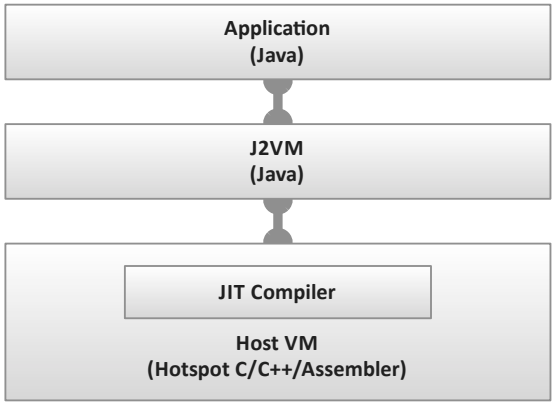


Figure 4.1: J^{2}VM is implemented in pure Java and executes JVM bytecode opcode by opcode. J^{2}VM runs on top of a *host VM*. The host VM can be interpreter-less (such as Jikes RVM).

J^{2}VM is implemented as a regular Java program that takes the name of the application to execute as command line argument. J^{2}VM then uses the class loader of the underlying host VM in combination with the BCEL [BCE] bytecode engineering library to load all class files referenced by the application program. Multithreaded programs are executed by creating one Java thread in the host VM running a copy of the interpreter for every thread the application starts.

J^{2}VM executes the application program by interpreting each bytecode instruction, starting with the `main` method of the application class. J^{2}VM maintains an internal data structure for the interpreter stack and local variables that is separate and distinct from the interpreter stack and local variables used by the host VM. All interpreted bytecode instructions operate on J^{2}VM's own data structures.

Simple bytecode instructions such as `iadd` are executed directly by J^{2}VM, i.e., by popping two integer values off the internal stack data structure, adding them, and pushing the result back onto the J^{2}VM internal stack. Both stack area and local variables are organized as one coherent array of *slots*. Each slot holds exactly one scalar data type (*integer*, *float*, *long*, *double*, *reference*), whereas *long* and *double* values occupy two consecutive slots. Method frames are allocated from this consecutive array by first reserving the number of local variable slots required by the method, and then using the remainder of the array as stack area. When a method is called, the JVM specification prescribes that arguments for the invoked method be pushed on the stack. We use this fact to propagate the arguments to the invoked method without actually copying data. For this we overlay the argument area of the stack of the caller with the local variable area of the callee. The slots pushed onto the stack become the argument part of the local variable area of the callee. The stack for the callee will then be allocated behind this new local variable area.

J^{2}VM supports throwing and catching exceptions. Exceptions are not thrown by our interpreter code but are instead the result of the host VM raising an exception while executing the implementation of the bytecode instruction being executed at that moment. When reading from an array, for example, we pop the array object reference and the index value from the internal stack and then pass them unchecked to the reflection interface of the host

VM to execute the array access operation. If the index is out of bounds, the host VM will raise an exception, and thus making it redundant for us to check for such error conditions.

The interpreter loop of J^{2}VM is surrounded by a catch block that intercepts all exceptions and then uses the internal program counter to determine which instruction triggered the exception, and the exception handler location to branch to. The interpreter then resumes at the address of the appropriate exception handler inside the method bytecode. If no local handler is available, J^{2}VM pops the method frame from the internal stack and re-throws the exception in the surrounding scope.

Native methods cannot be executed directly by our interpreter. Instead, J^{2}VM copies the arguments from its internal stack and invokes the requested native method via the reflection interface of the host VM.

Optimizing the Interpreter

While designing the J^{2}VM system we attempted various optimization techniques in order to improve interpretation performance. We focused our optimization efforts on several areas:

- We explored several ways in which the interpreter's internal execution stack can be expressed in Java in a way that incurs as little overhead as possible. To reduce the number of array bounds checks when operating on the internal execution stack, we use a doubly linked list of cells that is backed by an array. At runtime we maintain a reference that points to the top of the stack. When popping elements off the stack, we follow the linked-list pointers of the top cell instead of accessing cells via the array, and thus avoid any array bounds checks. If we need random access to the stack, we can simply index it using the array.
- We have experimented with various instruction dispatch techniques and found that the simple `switch` mechanism is most efficient, provided that the underlying host VM is able to compile the entire interpreter loop. This may not be the case if the interpreter loop becomes too large and compiler heuristics in the underlying host VM decide to ignore it, leading to double interpretation.
- Due to the high instruction dispatch latency in the interpreter, we attempt to collapse frequent sequences of instructions into single macro instructions. This helps eliminate some of the cost associated with instruction dispatch.

Evaluation

The results for section 2 and 3 of the Java Grande 2 [MCH99] benchmark are shown in figure 4.2 on the following page. For most benchmark programs our interpreter is within factor 4-9× of the speed of Hotspot's C/C++ and assembly language implementation of the interpreter.

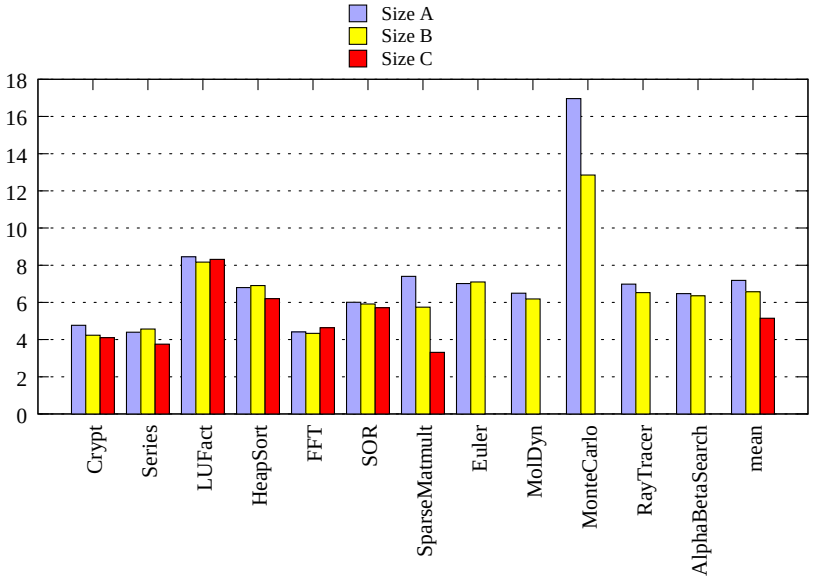


Figure 4.2: Performance of J^{2}VM on Java Grande section 2 and 3.

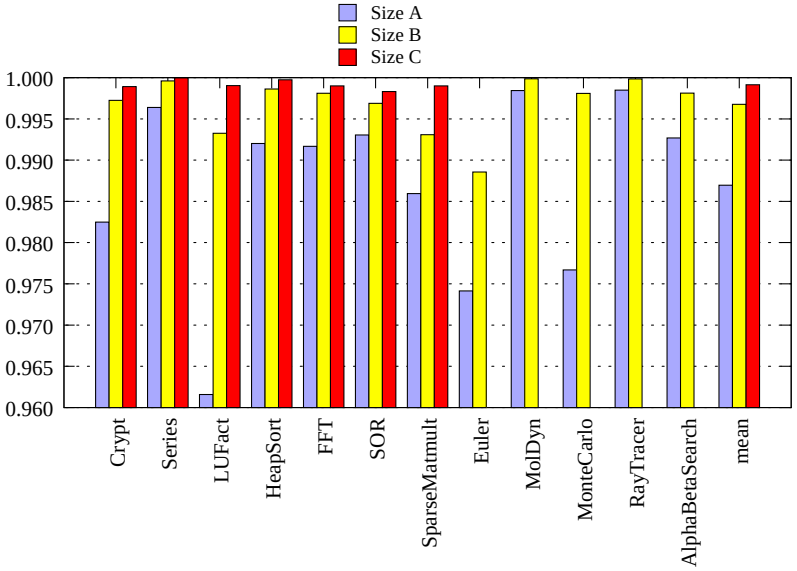


Figure 4.3: Proportion of time spent in compiled code for sections 2 and 3.

To evaluate whether the slowdown we have measured significantly impacts the overall performance of a mixed-mode virtual machine, we measured the fraction of the execution time that Hotspot spends interpreting bytecode vs directly executing Java code in compiled machine code form. [2]

[2]For this, we modified the Hotspot VM interpreter to count the number of bytecode instructions it interprets. We then ran the Java Grande 2 section 2 and section 3 benchmarks for all problem sizes (A, B, and C) first in mixed-mode, and then in interpretation only mode. For each mode we counted the number of instructions interpreted.

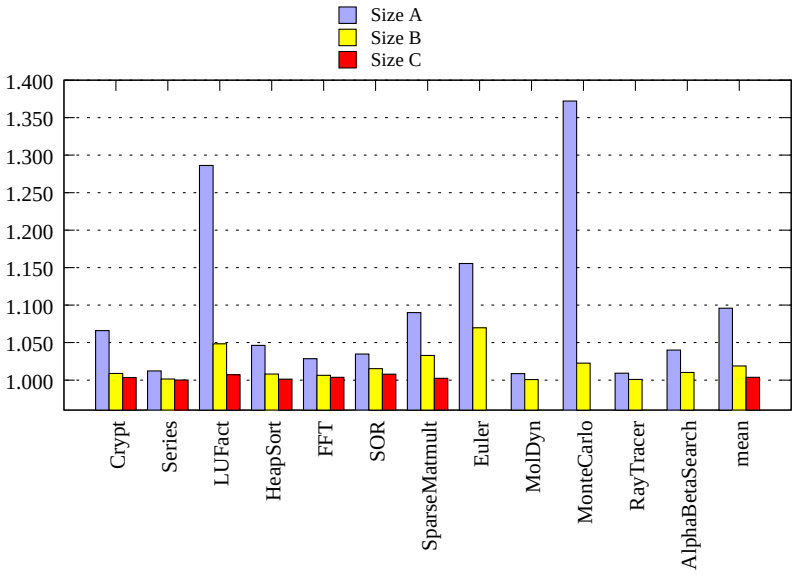


Figure 4.4: Simulated performance of J^{2}VM in a mixed-mode VM.

In mixed-mode, frequently executed code is compiled to machine code by the JIT compiler, and thus the VM *interprets* fewer bytecode instructions. The difference between the number of instructions interpreted in mixed-mode (with compilation) and interpretation mode (without compilation) can be used to estimate the fraction of runtime spent interpreting code vs running compiled code.

Figure 4.3 on the previous page shows the result of our analysis. Even for the smallest problem size (Size A), which runs for a few seconds only, at least 96% of the runtime is spent running compiled code. For larger problems (Size B) more than 99% of the runtime is spent in compiled code. For the largest problem size (Size C) the time spent interpreting instructions is essentially negligible. Our slower interpreter only affects 4% to 0.01% of the execution time that is not spent in compiled code. The overall slowdown of our system would be significantly smaller than 4 to 6 if J^{2}VM would directly interface with the JIT compiler and offload execution of "hot" code to the JIT compiler subsystem. While this is currently not the case in our prototype system running on top of Hotspot VM, this is merely an implementation limitation.

To estimate the performance of such a mixed-mode system that uses a pure Java-based JVM interpreter, we have applied the results of our analysis in figure 4.3 on the preceding page to simulate the execution time of the Java Grande 2 benchmark running on the Hotspot VM using Hotspot's compiled code performance, but with our 4-9× slower interpreter. The results are shown in figure 4.4. The slowdown of our interpreter does affect the results of the smallest problem size benchmark (Size A). For the larger problem sizes, on the other hand, the overall performance impact is negligible.

We discuss our optimizations in depth, as well as present a complete evaluation of the J^{2}VM system in [BGF07].

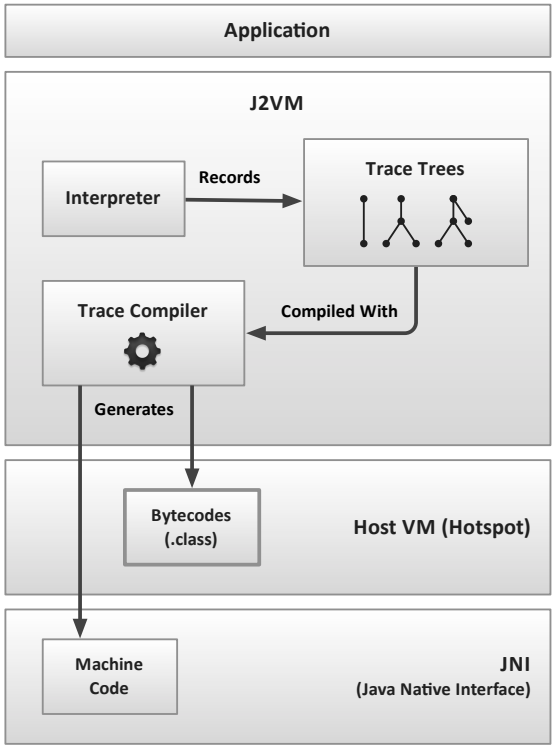


Figure 4.5: Trace-based compilation architecture in J^{2}VM.

4.2 Trace-Based Compilation

Our primary purpose for designing J^{2}VM was to establish an interpreter-based execution environment where we could easily experiment with trace-based just-in-time compilation. While the J^{2}VM system posed several advantages — garbage collection, type safety, easy debugging, and modern Java software development tools and IDEs accessibility — one significant limitation remained. It was unable to dynamically load, link and execute arbitrary machine code. The severity of this limitation led us to abandon J^{2}VM as a research platform in favor of Maxine, a fully meta-circular execution environment. In addition to the advantages it shared with J^{2}VM, Maxine has the added benefit of providing full control over how to load and link code, and how code interacts with the virtual machine. We present our experience with Maxine in chapter 5 on page 44.

Despite its drawbacks, research on J^{2}VM was influential in the development of the Mozilla TraceMonkey JavaScript VM, the Adobe Tamarin-Tracing ActionScript VM and our own future work on Maxpath. In this section we chronicle our experience with J^{2}VM.

The trace-based compiler in J^{2}VM uses trace trees to capture and compile frequently executed program regions. This process involves several phases:

4.2.1 Trace Anchor Detection

The first phase detects hot loop headers (anchors) using a simple backward branch detector built into the interpreter. The interpreter records the target of the branch whenever a backward branch is taken, and increments a counter associated with the branch target. Once a certain threshold is met, the backward branch target is considered a hot loop header, a strategy used in other state of the art Java trace-based systems [GPF06]. However, this turns out to be problematic, because not all backward branches are loop headers.

Moreover, significant problem arises when we identify hot loop headers this way. In Java, before class files are loaded, the bytecode verifier checks several type safety rules. In particular, it ensures that the *type state* (types of local variables and stack values) at each program location is fixed. This is done by using an iterative data-flow algorithm which is guaranteed to converge. The data-flow algorithm starts with the type state at the entry point of the method (the method signature) and propagates this type state through the control flow graph using abstract interpretation until reaching a fixed point. A method-based compiler operates in a similar fashion. It needs to know the type state at the method entry point so that it can assign types to live-in parameters and perform optimizations.

Likewise, a trace-based compiler also needs to know the type state at the beginning of the trace. The main difficulty is that the type state is not immediately known at the beginning of a trace. In J^{2}VM, we compute the entry type state during trace recording. We can infer the type of a local variable, or stack location by looking at the types of a load/store instruction's operands. JVML specifies two operations for accessing local variables: the $load_{i,t}$ and $store_{i,t}$ instructions, where t is the type of the operand being loaded or stored (one of `int`, `long`, `float`, `double`, `reference`) and i is the index of the local variable slot. During tracing, if we observe a $load_{i,t}$ we can deduce that the type of local i is t. If we encounter a $store_{i,t}$ we can deduce that the type of local i has now become t. This works well for linear traces, but breaks down as soon as we want to record additional traces and add them to the trace tree. Consider the example in figure 4.6 on the next page, both x and y variables are assigned the same Java local index `0` by the `javac` compiler, even though they are of differing types. The source compiler is free to reuse local `0` that was assigned to x because it is no longer live at the definition of y. If we trace the inner `while` loop, we discover that the type of local `0` is `INT`. If we record an additional trace after the inner loop condition fails, we trace the code following the inner loop. This trace may or may not lead to the inner loop again, depending on the condition of the outer loop. If the trace leads back to the inner loop, then — although local `0` becomes a `FLOAT` after the inner loop — it becomes an `INT` again before reaching inner loop. If the outer loop condition fails, the trace exits the code block and we incorrectly deduce the type of the local `0` to be, in fact, a `FLOAT`.

The fundamental problem is that we attempt to reason about the liveness of locals and without having a whole static control flow graph to work with. We infer liveness from execution traces, which is only an approximation. We solved this problem in J^{2}VM, by only computing liveness information from traces that eventually make their way back to the loop header. This solution does not require a more complicated liveness analysis and is easy to implement. Nevertheless, it is inflexible.

In chapter chapter 5 on page 44, we explain how we eventually solved these problems in Maxpath using a static analysis phase during bytecode verification. We augment the bytecode verifier so that it preserves type state information at select program points which we find interesting.

```
1 while (...) {
2     int x = 0;
3     while (...) {
4         ... = x;
5     }
6     float y = 1;
7 }
```

Figure 4.6: Java example where a local variable is reused with two different types.

4.2.2 Trace Tree Recording

In J^{2}VM, we monitored the execution of every transfer point. Whenever a transfer occurred we could optionally trigger the recording of an additional trace. If this trace led back to the anchor, we attached it to the transfer point, or otherwise discarded it. Each time a trace was successfully attached, the entire trace tree was recompiled. This is very different from previous trace-based compilation techniques because it allows for optimizations and register allocation across trace join points. The disadvantage of this approach is that we recompile the code repeatedly every time a new trace is attached. This has an unfortunate $O(n^2)$ asymptotic compilation complexity. We managed this problem by relying on a fast, linear-pass compiler, and limiting the number of traces in the tree to 8.

For compiling traces, we built a stream-based optimizing compiler that used SSA form as an intermediate representation. One of the advantages of trace-based compilation remains in how it simplifies the construction of SSA form. In SSA form, ϕ functions are used whenever data-flow is joined and their placement is non-trivial. It is easy to construct SSA form for trace trees because they have only one data-flow join point at the beginning of the loop header.

The work on our stream-based optimizing compiler was published at the International Conference on Objects, Models, Components, Patterns, Zurich, 2009 [BCGF09].

4.2.3 Trace Specific Compiler Optimizations

Aside from basic compiler optimizations, our trace-compiler featured several notable trace specific optimizations:

Guard Elimination proved to be a very important optimization on trace trees. During trace recording many conditions need to be guarded against. Explicit control flow checks,

type checks, virtual calls, and many implicit exception conditions require guards. Fortunately, many of these guards are redundant.

The lack of control flow join nodes in a trace tree means that if a guard condition is true, then it remains true for the remainder of the trace. We can use this observation to eliminate many successor guards whose conditions are subsumed by a dominating guard. For instance, consider a sequence of guards $\{g_0 : x < 2, g_1 : x < 4, g_2 : x < 8\}$, the first guard g_0 subsumes both g_1 and g_2, and we can be safely eliminate them. This optimization can be applied to general CFGs as well, albeit less effectively because of join nodes. What is uniquely specific to trace-based compilers is its capacity to rewrite dominating guards to become more constraining. Doing so, allows us to eliminate guards even if they are not subsumed by dominating guards. Consider the following reversed sequence of guards $\{g_0 : x < 8$, $g_1 : x < 4$, $g_2 : x < 2\}$. Here we cannot eliminate any guards because no dominating guard subsumes a dominated guard. We can however use a clever trick. We can rewrite the first guard to read, $x < 2$. The sequence of guards now becomes $\{g_0 : x < 2$, $g_1 : x < 4$, $g_2 : x < 2\}$. The two remaining guards are now redundant and we can eliminate them. Rewriting guards to become more constraining is a safe operation in a trace-compiler. If $x >= 2$, then the first guard fails and execution is resumed in the interpreter, where the guard condition is re-evaluated. In the interpreter, the condition will pass and execution will resume correctly.

Our experience with guard elimination in J^{2}VM led us to explore type specialization optimizations in dynamic languages. Many dynamic languages have bytecode operations that are only polymorphic by definition but mostly monomorphic during runtime. These operations often involve a large number of type checks before performing an actual operation. For instance, a simple `ADD` operation may mean several things depending on the types of its operands. If both of its operands are numbers, then we can simply add them together. Otherwise, if the left operand is a string, then we can convert the right operand to a string and concatenate it to the left operand. Moreover, if the left operand is a number, and the right is a string, then we can convert the right operand to a number and add it to the left operand. The key to improving performance is to specialize the implementation of bytecode operations for the most common type configuration. This reduces the number of type checks performed and is sometimes accomplished by using polymorphic inline caches [HCU91] or instruction quickening [Bru10].

In order to explore trace-based compilation in the context of dynamic languages, we prototyped a JavaScript VM in Java and ran it on top of our J^{2}VM trace-based compilation framework. Using trace compilation, we recorded the execution of sequences of polymorphic JavaScript bytecode operations whose implementations were expressed in Java and optimized away redundant type guards. Our work in J^{2}VM inspired the design of the current trace-based compiler found in Mozilla Firefox, which is used by over 500 million users every day. Moreover our work also inspired the SPUR Project [BBF$^+$10] at Microsoft Research, where a JavaScript VM was built on top of a trace-compiler for CLR [MG01]. We detail our work on SPUR in chapter 7 on page 91.

Partially Invariant Code Motion is an important optimization on traces because it allows us to hoist invariant code outside of loops. This can improve performance considerably

because hoisted instructions are executed only once instead of each loop iteration. Invariant code is code which is side-effect free and independent of the loop induction variable (variable that gets incremented or decremented with each loop iteration). In most loops, little code is invariant since experienced programmers routinely hoist trivially invariant code manually, thus leaving the compiler with little work to do. In more complicated situations, the compiler often fails to identify invariant code because it must analyze all code paths through loops. However, code may be *partially invariant*, which motivates the following idea:

> *Code is partially invariant if it is invariant along some code paths and not others. By using trace-based compilation, we can focus on frequently executed paths, and ignore alternate code paths that may cause code to become not invariant.*

Consider the simple loop example in figure 4.7a on the following page. A while loop repeatedly appends the string `''Hello World''` to an `ArrayList<String>` using the `add()` method. Occasionally, the `add()` method needs to resize the array store that backs the array list itself. Even after inlining, a conventional compiler cannot hoist out frequent operations on the `elementData` array because of this seldom occurring resizing operation. If the `elementData` array is not large enough to accommodate additional elements, the `ensureCapacity` method creates a new larger array and assigns a new value to `elementData`. This means that the `elementData` variable is partially invariant. Luckily, on the most frequently executed path, it is invariant. A trace-based compiler would likely only ever see and compile the frequently executed path, and ignore the path that causes `elementData` to become not invariant. Once `elementData` is provably partially invariant, the trace-based compiler can hoist all null checks, and array abounds checks related to `elementData`.

Code hoisting presents a challenge to our incremental trace tree compilation technique. We may invalidate assumptions made about code invariance as we accumulate additional traces. For instance, in the previous example, if we were to record the additional trace covering the case where the `elementData` array is extended, we would invalidate the assumption about its invariance. After adding traces to a trace tree, we must always perform invariant code motion on a fresh tree instead of one that has already been subjected to invariant code motion. In J^{2}VM, we preserve an original, non-optimized version of the trace tree as we extend it. The first phase of the compilation process clones the tree and streams it through the compilation pipeline. This way, we never change the original tree during optimizations. The details of this approach are presented in [BCGF09].

Phase Detection Dynamic compilers can optimize application code specifically for observed code behavior. Such behavior does not have to be stable across the entire program execution in order to be beneficial for optimizations. It needs to be stable only for a certain program *phase*. To specialize code for a program phase, it is necessary to detect when the execution behavior of the program changes (*phase change*). Trace-based compilation is an efficient method for detecting such phase changes. A trace tree is a collection of frequently executed code paths through a code region, which is assembled dynamically at runtime as the program executes. Program execution tends to remain within such a trace tree during a stable phase, whereas phase changes cause a sudden increase in transfers from the trace tree. Because trace trees are recorded at runtime by observing the interpreter, the actual

values of variables and expressions are also available. This allows a definition of phases based not only on recurring control flow, but also on recurring data values. The compiler can use constant values for variables that change their value rarely and rely on phase detection to handle the case when the variable value actually changes. Our evaluation shows that phase detection based on trace trees results in phases that match the intuitive expectation of a programmer and are also useful for compiler optimizations. We present our findings in [WCB+09].

```
1 ArrayList<String>
2   list = new ArrayList();
3
4 while (...) {
5     list.add("Hello World");
6 }
```

(a) Simple loop that adds a string to an `ArrayList<String>`.

```
1  public boolean add(String e) {
2    ensureCapacity(size + 1);
3    elementData[size ++] = e;
4    return true;
5  }
6
7  public void ensureCapacity(int minCapacity) {
8    modCount++;
9    int oldCapacity = elementData.length;
10   if (minCapacity > oldCapacity) {
11     Object oldData[] = elementData;
12     int newCapacity = (oldCapacity * 3) / 2 + 1;
13     if (newCapacity < minCapacity) {
14       newCapacity = minCapacity;
15       elementData =
16         Arrays.copyOf(elementData, newCapacity);
17     }
18   }
19 }
```

(b) ArrayList `add()` and `ensureCapacity()` methods.

Figure 4.7: Simple loop which adds a string to an `ArrayList<String>` showcasing how partially invariant code motion can be used to hoist loop invariant code.

4.2.4 Code Loading, Linking & Execution

Loading, linking and executing compiled traces in J^{2}VM proved to be a challenge. The difficulty lies in the ability to load and execute unsafe code within the Java virtual machine. For safety reasons, JVMs are designed to prevent unsafe code injection. To circumvent this security policy we attempted to load code in two ways: through bytecode generation and dynamic class loading, and through the Java Native Interface (JNI).

Dynamic Class Loading

The Java virtual machine provides a facility for dynamically loading code at runtime. The usual procedure is to manually assemble JVML bytecode into a `.class` file and then define a custom class loader that loads it into the VM. The code is subsequently compiled and linked by the JVM's compiler. Compiling traces to `.class` files has several interesting properties:

- Because we use the underlying JVM's optimizing compiler, the trace compiler does not have to focus on low-level compiler optimizations. Bytecode and standard control flow optimizations, register allocation, and machine code generation are all handled by the host VM's optimizing compiler. This allows us to focus on optimizations that are not usually performed by the underlying compiler, such as partially invariant code motion, guard elimination, etc. Additionally, this reduces the trusted code base, and most importantly allows us to treat the compiler backend as a control variable during experimentation.

- Generated `.class` files can either be executed by the host JVM directly, or yet again through J^{2}VM. When executing under the host JVM, trace trees are subject to the host JVM's adaptive compilation scheme. Usually, the compiled trace tree is executed and profiled by the host JVM's interpreter until a heuristic triggers its recompilation by the optimizing compiler. Without making changes to the internals of host VM, it is not possible to hint that some dynamically loaded code should be immediately compiled by the optimizing compiler. Consequently, we always incur an interpretation overhead when executing compiled trace trees.

  When executing under J^{2}VM, compiled trace trees are again subject to trace compilation and optimization until eventually some form of the trace tree is executed by the host JVM directly.

- Before loading a class, the Java bytecode verifier first checks several bytecode properties. For instance, in Java, class member methods cannot access private methods or fields of sibling classes, and final fields can only be assigned in class constructors.

  A trace tree however may capture program flow that reads/writes private or final fields, or even invokes private member methods. A generated `.class` file that performs the same operations fails verification, because as a sibling class it does not have access to the private members accessed along the recorded trace tree.

  Conveniently, the Java platform provides an introspection mechanism called the Java Reflection API that allows programmers to access arbitrary class fields, invoke methods and create objects. We tried using this API to gain access to private members, by inlining calls to the Java Reflection API whenever a field access was required. Figure 4.8 on the next page shows how field accesses and method invocations can be expressed through the Java Reflection API.

  Although modern JVMs provide some optimizations for code that uses the Reflection API in many cases it fall short. For highly reflective code, the host JVM's optimizing compiler fails to perform many optimizations because it has a hard time reasoning about variable read/write access through the Java Reflection API. Compiling code this way was very inefficient and led us to abandon the Reflection API and instead explore an alternate approach.

  Luckily, using a loophole [3] in the implementation of the HotSpot JVM we managed to load code that bypassed the Java bytecode verifier, thus giving us direct access to private class members. This allowed us to compile arbitrary traces to `.class` files (such as in figure 4.8a on the following page), and have them compiled and executed by the host JVM.

[3] HotSpot's bytecode verifier skips all classes subclassed from an internally defined class. It uses this mechanism in order to implements parts of the VM in Java that require unsafe access.

```
1 Example example = new Example();
2 example.privateIntField += 42;
3 example.privateMethod(42);
```

(a) A compiled trace which accesses private members and which does not pass Java bytecode verification.

```
1 Example example = new Example();
2 Field field = Example.class.getField("privateIntField");
3 field.setAccessible(true);
4 field.setInt(example, field.getInt(object) + 42);
5
6 Method method = Example.class.getMethod("privateMethod", int.class);
7 method.invoke(example, 42);
```

(b) A compiled trace that uses the Java Reflection API to gain access to private members and which passes Java bytecode verification.

Figure 4.8: Using the Java reflection API to access private class members.

Compiling trace trees to pure Java `.class` files without modifying the underlying host JVM proved to be impractical. In future work on trace compilation we provisioned special VM support for compiled traces.

- We compile trace trees by generating Java `.class` files with a single method (trace tree method). This method can become very large, especially if the traces are long. Many JVMs are tuned to simply ignore very large methods. These methods are not optimized and are usually executing only with the interpreter.

- Executing a trace tree involves two steps: first all live variables that are used in the trace are passed to the compiled trace, and second, execution is transferred to the compiled trace tree method. Since the type state is known at trace entry points (see section 4.2.1 on page 32) we can pass all live variables either by-reference by passing in references to J^{2}VM slot objects containing the live-in values, or by-value by forcing the signature of the trace tree method to match the type state at the trace entry point. Call by-reference is more effective because it allows the trace tree method to update the referenced slots at trace exit points. Although arguments are passed by-reference, we can cache them in local variables so that the host JVM's register allocator can eventually assign them real machine registers and avoid repeatedly reading their values from J^{2}VM slots. Figure 4.9a on page 40 shows a simple Java program loop along with its compiled trace tree method in figure 4.9b on page 40. The J^{2}VM interpreter calls the trace tree method and passes the two slot objects containing the value of `j` and `i`. The trace tree method then caches the slot values `js` and `is` in local variables `j` and `i` respectively.

- Reconstructing the interpreter state at trace exits involves coping all live-out variables back to the interpreter stack slots stored in J^{2}VM frames, and resuming execution in the interpreter. A trace tree may have many exit points each with different type states. At every trace exit point, we generate bytecode that updates the slots in J^{2}VM frames with the values stored in cached local variables. The example in figure 4.9b on page 40 has two trace exit points. Code is explicitly generated to update the values of slot `js` and slot `is` at each of these exits before finally updating the program counter

of the interpreter and returning.

Although not explicitly shown in the example in figure 4.9 on the next page the situation is further complicated by function inlining. If a guard appears in an inlined method, the corresponding trace exit code must be able to create additional J^{2}VM stack frames and update their slots as well. Since these slots are not passed in as parameters, the exit code must be able to infer them from the reference to the J^{2}VM interpreter that is passed in. As may already be apparent from glancing at the code in figure 4.9b on the following page the size of trace exit code can quickly dwarf the rest of the trace tree method code. In practice, we have observed that roughly 80% of the generated code is devoted to trace exits which is problematic because it eventually inhibits the host JVM's optimizations.

In order to reduce the size of trace exit code we considered exploiting some of the redundancies apparent in trace exit code. As can be observed in figure 4.9b on the next page, the trace exit code is nearly identical in both cases, exception being the last statement which resumes execution either at `L0` or `L1`. We can compress the code by factoring out the identical code into a Java subroutine and using the `JSR` and `RET` JVML bytecode to call the subroutine and return from it. Java subroutines are used in a similar fashion in Java to succinctly compile the `try-finally` language construct. The `finally` block needs to be executed whenever control exits the `try` block which may happen as a result of normal exit of the try block, or as a result of an exception being caught. Instead of duplicating the code in the `finally` block at each of these exit points, a subroutine is used.

Computing a set of subroutines that cover all trace exit code can be accomplished using a prefix tree. A prefix tree can be used to encode maximal subsequences of shared trace exit code and arrange them in a way that minimizes code duplication. Each node in the prefix tree becomes a subroutine, and each trace exit code segment is reduced to a sequence of subroutine calls. Traversing the tree backwards, from the leaves to the root and emitting JSRs and RETs along the way can compress trace exit code. Although an academically novel approach, in practice, using subroutines in Java comes with a substantial cost, as is presented in [Fre98]. Java subroutines significantly increase the complexity of bytecode verification and compilation, something which we were trying to avoid in the first place.

Another possibility is to factor out frequently occurring code snippets as methods. The difficulty with this approach is that we have to pass live-out variables to these methods as parameters so they can correctly update the J^{2}VM interpreter state.

In the end, compressing trace exit code is overly complex and not warranted especially since it may slow down execution due to the overhead of subroutines and method calls. In future work we tackled this problem in two different ways: first we reduced the number of guard instructions by covering more program code using trace regions, and second we generated trace exit code on demand, as needed, rather than ahead of time as was done in the J^{2}VM system.

```
1  program() {
2      int j = 0;
3  L0: for (int i = 0; i < 100; i++) {
4          j += i;
5          if (j == 15) {
6  L1:         j *= 2;
7          }
8      }
9      System.out.println("Sum:" + j);
10 }
```

(a) A program whose execution is traced and compiled on the right.

```
1  trace(Interpreter interpreter,
2        Slot js,
3        Slot is) {
4
5      // caching live-in parameters
6      int j = js.intValue;
7      int i = is.intValue;
8
9      while (true) {
10         if (i >= 100) { // guard i < 100
11             // trace exit to L0
12             js.intValue = j;
13             is.intValue = i;
14             interpreter.resumeAt(L0);
15             return;
16         }
17
18         if (j == 15) { // guard j != 15
19             // trace exit to L1
20             js.intValue = j;
21             is.intValue = i;
22             interpreter.resumeAt(L1);
23             return;
24         }
25
26         sum += i;
27     }
28
29 }
```

(b) A compiled trace tree showing how control is transferred to and from compiled trace tree methods.

Figure 4.9: Compiled trace tree.

Java Native Interface

As an alternative to dynamic class loading, we explored using the Java Native Interface (JNI) to load and execute dynamically generated code. JNI is a programming framework that allows Java programmers to make use of native code written in other languages, and interface with platform-specific features and other system libraries that are not exposed through the standard Java class library. JNI allows native code to access Java features as well, such as: creating objects, accessing fields and arrays, invoking Java methods. Although these operations are provided by JNI through a safe mechanism that ensures the internal state of the JVM is uncompromised, there are still many opportunities (hacks) to bypass safety guarantees.

The example in figure 4.10 on page 43 shows how JNI can be used to execute arbitrary code. At runtime, the Java method "`public static native void execute(byte[] code)`" is linked to the "`Java_Example_execute(JNIEnv *env, jbyteArray codeArr)`" native method in the JNI native library. The latter converts the `byte[]` code buffer to a `char *` using the `GetByteArrayElements()` JNI method. This is necessary because JNI code cannot access JVM objects directly since object layout is only known to the JVM and not exposed to JNI code. Since JNI libraries can be loaded and executed by any standard JVM, hard coding object layout information in native code would thus inhibit portability.

Additionally, garbage collection also places a constraint on JNI code. Because the JVM is free to move objects around in memory during a garbage collection (GC), JNI code cannot store and manipulate direct reference to Java objects. Instead, JNI uses one level of indirection through reference handles which are maintained by the JVM and updated accordingly whenever objects move around in memory.

As part of the J^{2}VM system we also built an optimizing compiler, a register allocator and assembler that directly generated x86-64 code, instead of relying on JVML bytecode as a backend. Having complete control over how code is generated, linked and executed is important in VM design, and this was not possible with our previous dynamic class loading approach. JNI provides an escape hatch from the JVM and can be used to dynamically load and execute machine code, however with several caveats.

JNI is designed to abstract the internal details of the host JVM and is in stark contrast with the goals of our trace compiler. Dynamically generated trace code requires efficient access to object fields, arrays, object allocation and unrestricted access to other low level VM services such as garbage collection, exception handling, stack walking, etc.

For safety reasons, JNI purposely attempts to hide these services away from JNI code. For instance, in figure 4.10 on page 43 the `GetByteArrayElements()` returns a `char *` pointer to a byte array managed by the JVM. If a garbage collection were to occur after a call to this method, and the byte array were to be relocated to another area in the heap, the `codePtr` would never be updated to reflect the new location, which would result in a VM crash. Within Java code, the GC is responsible for patching references to moved objects. This can be accomplished with the help of the optimizing compiler which produces detailed meta-data at specific program locations called *safepoints*, information the GC can use to identify exactly which memory locations or registers hold object references that need to be patched. Native code is compiled by compilers that do not produce this type of meta-data and thus the code does not support reference patching. Instead, JNI uses a combination of *object pinning*, *copying* or *object handles* as a solution. Because the `GetByteArrayElements()` function returns a direct `char*` pointer to the code buffer, it must either *pin* the code array in memory temporarily preventing it from being moved by the GC, or make a temporary local copy of the code array and later update its contents when control is handed back the the JVM.

JNI is designed to interface with legacy code, or to gain access to unavailable system resources, it is not designed as a way of factoring out critical sections of Java code into native C/C++ code with the hope of achieving better performance.

In our J^{2}VM trace-based compiler, JNI provided us with a convenient and efficient way of loading and executing code at runtime, but severely limited us in terms of what that code could do. Because JNI functions that manipulate object fields, arrays, or invoke methods are so inefficient, we had to avoid them and violate the safety guarantees provided by JNI by gaining direct access JVM objects. Using intimate knowledge about how HotSpot implements the JNI interface on the VM side, we could gain unsafe access to JVM objects through direct references. Since we could only use these references between GC events, we disabled the GC whenever executing trace compiled code. This however, is very limiting, because GC is usually only triggered by new object allocations. Disabling the GC during

trace compiled code meant that no objects could be allocated, which is debilitating for Java code.

Many state of the art garbage collectors rely on write barriers [4] on updates to reference fields to improve GC performance. In HotSpot, the write barrier updates a global data structure called the *card table*. This data structure is a large contiguous bit vector mapping the entire Java heap space, each bit represents a page (card) in the heap of a certain size. Whenever code updates a reference field, the corresponding bit in the card table for the address of the updated field is toggled. This card table informs the GC which pages in the heap have observed updates, so that the GC need not waste time scanning objects that have not been modified since the last GC event. Although, the GC was sometimes disabled in our J^{2}VM trace-based compiler, we still needs to comply with the way HotSpot marked the card table. Using JNI it's difficult to gain easy access to the card table, and other VM specific internal details such as heap page sizes. We worked out these issues by looking at the HotSpot source code, and hard coding many details in our trace compiled code.

One advantage of directly generating machine code is that we could improve the execution of trace exits. Instead of generating code for the exits explicitly, we encoded the location of local variables at the exit point in an auxiliary data structure. At the trace exit point, we invoked a routine that inspected this data structure and restored the state of J^{2}VM interpreter.

Conclusions

Dynamic class loading is a safe mechanism for loading additional code into the JVM but it's too restrictive for our use. The large amount of code required to adapt parameters and restore the state of the interpreter at trace exits and the inability have direct control over how and when generated code is optimized makes dynamic class loading inadequate for our use.

As an alternative we also explored generating native machine code ourselves and executing it through the Java Native Interface. This has the advantage of giving us full control over how code is generated and compiled, but it requires a lot of reverse engineering so that the generated code plays along nicely with the rest of the HotSpot system, if at all possible.

Each approach has its unique advantages and disadvantages, in future research we converged on a combination of these two approaches. In Maxpath, we built the trace compiler into the VM, rather than completely on top of it. Doing so allows us to have complete control over the internals of the VM. Instead of generating class files, we rely on a lower level, internal intermediate representation that can still be optimized by the compiler and can be manipulated at will. We describe in detail the design of our Maxpath system in chapter 5 on page 44.

[4] Not to be confused with processor write barriers.

```java
// Example.java
class Example {
   static {
       System.loadLibrary("nativeLibrary");
   }
   public static native void execute(byte[] code);
}

// Example.cpp (in nativeLibrary)
JNIEXPORT void JNICALL Java_Example_execute(JNIEnv *env, jbyteArray codeArr) {
   /* Get the Java byte array as a char buffer. */
   char *codePtr = env->GetByteArrayElements(codeArr, 0);

   /* Tell the Operating System to unprotect the code region. */
   mprotect(codePtr, 1024, PROT_EXEC | PROT_READ | PROT_WRITE);

   /* Execute code. */
   void (*code)() = codePtr;
   code();
}
```

Figure 4.10: Executing dynamically generated code through the Java Native Interface.

Chapter 5

Maxpath

Trace-Based Compilation in Meta-Circular Virtual Execution Environments without Interpreters

Trace compilation is well suited for interpreter-based execution environments because the control flow of an application program is highly visible and recordable. In this chapter, we demonstrate that trace compilation is also feasible and beneficial in virtual execution environments without interpreters where it is more difficult to monitor the control flow of an application.

We present the implementation of Maxpath, a trace-based Java just-in-time compiler for the Maxine virtual machine. Maxine uses a tiered compilation strategy where methods are first compiled with a baseline just-in-time compiler in order to collect profiling information, and then recompiled with an optimizing compiler for long-term efficient execution. We record traces by dynamically inserting instrumentation code in baseline methods. Execution traces are first collected into trace regions, after which they are compiled, optimized and linked to baseline methods for efficient execution. We show that trace-based compilation is an effective way to focus scarce compilation resources on performance critical application regions.

5.1 Introduction

In this chapter, we present the design and implementation of Maxpath, a complete trace-based just-in-time compiler for the Maxine VM [Mat08]. Unlike other VMs where trace-based compilers have been implemented, Maxine does not use an interpreter at all. Instead, it relies on a lightweight baseline just-in-time compiler for the initial execution of Java bytecode. The execution performance of this type of just-in-time compiler is relatively good, much faster than that of a traditional interpreter. The design of Maxine presents two challenges to trace-based compilation: 1) recording and controlling execution of a compiled code is significantly more difficult than in an interpreter, and 2) since the performance difference between the baseline baseline just-in-time and the optimizing compiler is smaller

than it is in a mixed-mode interpreted environment, the trace-based compiler needs to be more aggressive in finding compilation-worthy regions in order to pay back compilation cost or any incurred overhead.

Our research makes the following contributions:

- Identification and recording of frequently executed code traces using instrumentation in execution environments without interpreters.
- An efficient control transfer mechanism for entering and returning from traces.
- A design strategy for retrofitting a Java virtual machine with a traced based compiler.

5.2 Maxine VM

In order to understand the design of Maxpath it is first necessary to understand the design of the Maxine VM. The following section describes several architectural decisions in the Maxine VM that are pertinent to Maxpath. Maxine is a meta-circular research VM under development at Oracle Labs (previously Sun Labs). It is written entirely in the Java programming language, leveraging the benefits of meta-circular design and the state-of-the-art development tools.

5.2.1 Meta-Circular Design

Maxine is meta-circular! It is implemented in the same language that it executes. Moreover, it relies on the JDK (Java Development Kit) which is also implemented in Java. Portions of the JDK which are traditionally implemented in lower level languages, because they cannot be effectively expressed in Java, are also implemented in Java inside of Maxine. Figure 5.1 on the next page depicts the structure of Maxine.

In conventional design, the VM is implemented in a lower level language such as C/C++. This creates a hard semantic boundary between the language the VM is implemented in and the language the VM implements. Java compiler optimizations cannot cross this boundary because they cannot understand C/C++ code. Similarly, C/C++ optimizations cannot cross this boundary because they cannot understand Java code. Therefore, the VM and the application it executes cannot be optimized together.

In meta-circular design, the VM is implemented in Java and runs on top of itself along with the application it executes. For example, Maxine's optimizing compiler can self re-optimize while adapting to the execution workload an application it's running. Meta-circular design has a lot of implications that are not immediately obvious. For instance, the garbage collector cannot freely allocate memory since it would have to collect itself. Also, the code for handling exceptions cannot throw exceptions because it would have to handle its own exceptions. In short, meta-circularity is a breeding ground for self-referential notions and

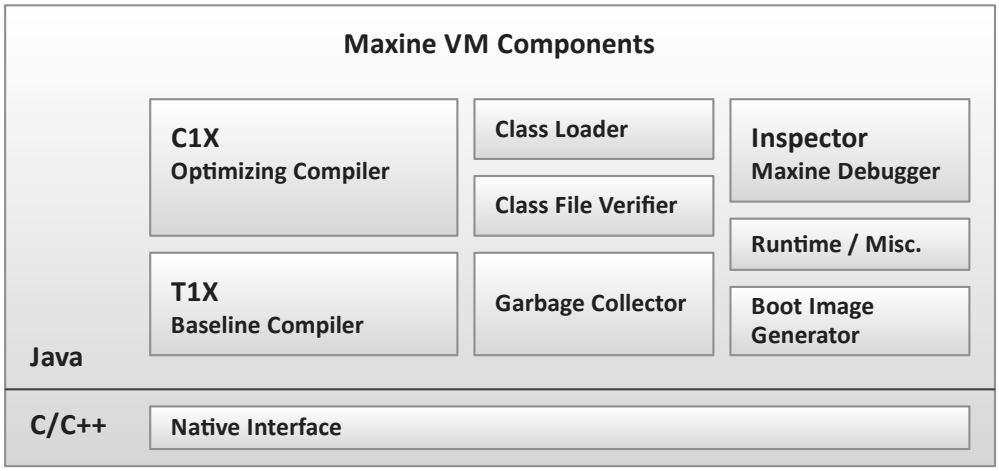
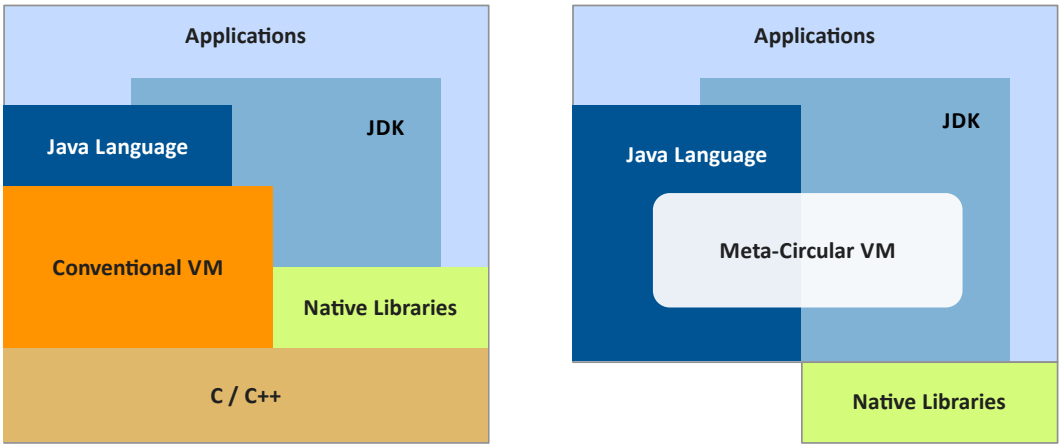


Figure 5.1: Maxine VM components, consisting largely of Java code, and a thin layer of C/C++ that interfaces with the operating system.

a lot of effort is expended trying to prevent cyclic dependencies at runtime. Figure 5.2 contrasts meta-circular VM design (b) with conventional VM design (a).



(a) Conventional VM design.

(b) Meta-circular VM design; written in the same language that it executes.

Figure 5.2: Conventional VM design vs. Meta-Circular VM design. Figure source Mathiske et al. [Mat08].

5.2.2 Bootstrapping & Execution

Bootstrapping is an intricate process where Maxine first runs on top of a host VM in order to compile itself, and then transitions its execution so that it runs on top of itself. Figure 5.3 on page 48 and figure 5.4 on page 49 depicts some the following steps in the bootstrapping process:

1. *Compilation*: Maxine must compile a minimal subset of its source base. It does this by computing a transitive closure R from the VM's entry point, meaning it compiles all methods that are reachable form this point. Unfortunately, this is a very large set since it's likely to include the majority of the JDK. For instance, the transitive closure of a simple simple Java statement such as `System.out.println(‘‘Hello World!’’)` is likely to use dozens of classes from the JDK related to security, networking, I/O, regular expressions and more [WGF08]. Therefore, Maxine prunes this transitive closure by constructing a set $P \subset R$ such that a method $m \in R$ is in P iff m is not compilable at runtime using the methods in P. In other words, Maxine tries defer the compilation of methods in R until runtime if these methods are compilable with the bootstrapped VM. The set P is built empirically and not by any verifiable process.

2. *Heap Transplant*: The bootstrapping process activates several Maxine subsystems, such as class loading, verification, reflection, compilation, heap management, etc. Since these subsystems are all implemented in Java, their classes are loaded and initialized during the bootstrapping process. Once Maxine is bootstrapped and executes code without the assistance of a host VM, its subsystems must appear as though they have already been pre-initialized. For instance, static class variables that are part of the VM are initialized during boot strapping, and not during execution. Maxine performs this trick by computing yet another transitive closure but on the object graph. Using Java reflection, Maxine can introspect the state of its heap in the host VM and rewrite it into a form that is understandable later when Maxine starts up which is necessary because the host VM and Maxine use different heap layouts.

3. *Execution*: Maxine includes a very small layer of C/C++ code (*Native Interface*) that is responsible for interacting with the operating system. This small layer is also responsible starting up the VM. The startup code maps the generated binary code image and the associated heap image into its address space and transfers control to the VM's entrypoint method which is implemented in Java.

5.2.3 Baseline & Optimizing Compilers

Maxine uses a tiered compilation strategy where methods are first compiled with a baseline compiler in order to collect profiling information, and then recompiled with an optimizing compiler for long-term efficient execution. The baseline compiler in Maxine is essentially an inline-threaded interpreter. Each Java bytecode is translated to a snippet of pre-assembled machine instructions that operate on the machine stack. The baseline compiler generates code quickly, in a single forward pass, by concatenating and linking these snippets together. An important aspect of the Maxine baseline compiler is that it preserves the Java bytecode stack semantics because it greatly simplifies the design of the compiler and the VM. Figure 5.5 on page 49 contrasts the output of the baseline compiler (a) with the optimizing compiler (b) for the Java statement `b = a - (1 + d)` which is compiled to the following sequence of bytecode: `ILOAD_0` (load variable a), `ICONST_1`, `ILOAD_3` (load variable d), `IADD`, `ISUB`, `ISTORE_1` (store into variable b).

The optimizing compiler [TWSC10] is slower than the baseline compiler but produces better

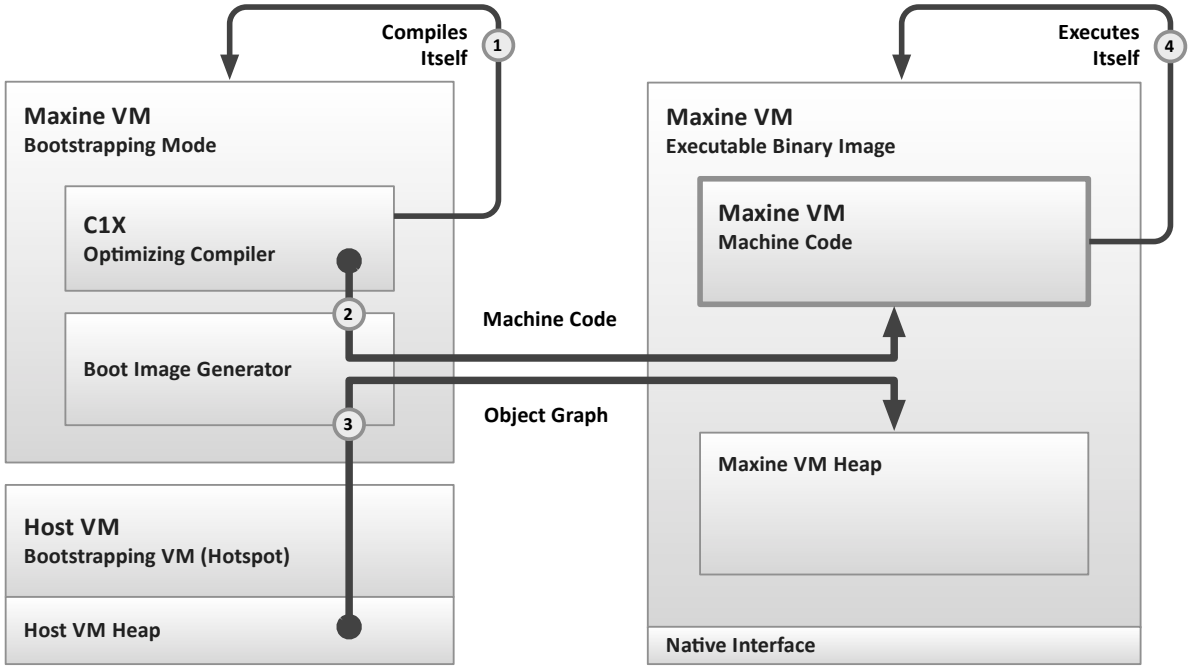


Figure 5.3: Bootstrapping process. Maxine runs on top of a host VM such as Hotspot and compiles itself using the C1X optimizing compiler (1). The boot image generator then computes a transitive closure on the Maxine's object graph and transplants it (using Java reflection) into a form understood by Maxine (2, 3). Code, data and a native interface are all linked together, completing the bootstrap process. Maxine is now able to execute independently of the host VM (4).

code. It performs various standard optimizations, such as method inlining, and register allocation.

5.2.4 Stack Frame Layouts & Adapter Frames

The optimizing compiler uses a stack frame layout (figure 5.6b on page 50) and a calling convention (AMD64) [MMM10] which is incompatible with the stack frame layout used by the baseline compiler (figure 5.6a on page 50). This requires the use of adapter frames whenever a method compiled with the baseline compiler invokes a method compiled with the optimizing compiler. The same is true the other way around. Adapter frames are machine code stubs that are used to adapt between different calling conventions. There are several types of adapter frames, one for each stack frame layout transition:

- Baseline → Optimized: the first few arguments are copied from the Java stack into machine registers, while the remaining arguments are overflowed on the stack as expected by a callee compiled with the optimizing compiler.
- Optimized → Baseline: arguments are pushed onto the stack as expected by a callee compiled with the baseline compiler.

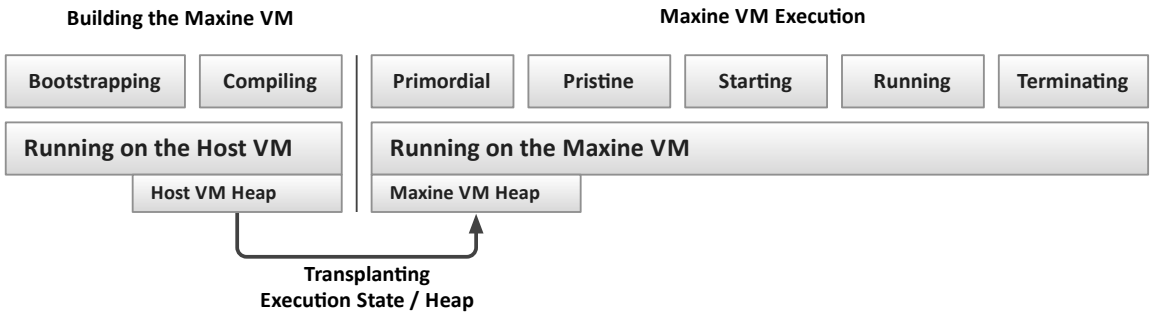


Figure 5.4: Maxine execution phases, from left to right: The *Bootstrapping* and *Compiling* phases run on top of the host VM and are responsible for building the Maxine VM executable image. The remaining phases run on top of the Maxine VM. The *Primordial*, *Pristine* and *Starting* phases are responsible for bringing up several subsystems such as threading support and synchronization. Once the VM reaches the *Running* phase, it's ready to execute Java applications. Finally the *Terminating* phase is responsible shutting down the VM gracefully.

```
1  ILOAD_0:
2    push [RBP - 0]
3  ICONST_1:
4    push 1
5  ILOAD_3:
6    push [RBP - 24] // 3 x 8 = 24
7  IADD:
8    pop  RBX
9    pop  RAX
10   add  RAX, RBX
11   push RAX
12 ISUB:
13   pop  RBX
14   pop  RAX
15   sub  RAX, RBX
16   push RAX
17 ISTORE_1:
18   pop  [RBP - 8] // 1 x 8 = 8
```

(a) Machine code generated code by the baseline compiler. `RBP` is used as a frame pointer to access frame locals.

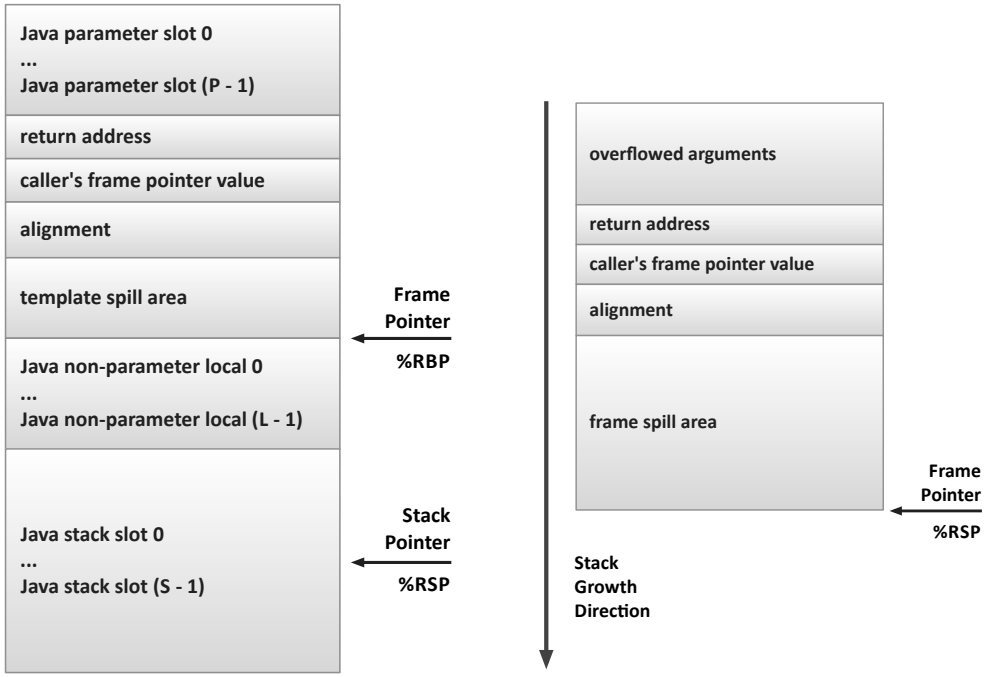
```
1 mov  rcx, rdx
2 add  rcx, 1
3 mov  rbx, rax
4 sub  rbx, rcx
```

(b) Machine code generated code by the optimizing compiler assuming variable a, b and d are allocated the RAX, RBX, and RDX machine registers respectively, and that RCX is a scratch register.

Figure 5.5: Contrast between the baseline (a) and optimizing (b) compiler for a short program. The code produced by the baseline compiler emulates the Java stack on the machine stack and is significantly less efficient than the code produced by the optimizing compiler.

- Baseline → Region: space is reserved on the stack for the deepest possible transfer, but no arguments are copied. Section 5.3.5 on page 68 describes this type of adapter frame in more detail.

Adapter frames are shared by methods with the same signature. Every compiled method has two entry points: a baseline entry point and an optimized entry point. If an optimized



(a) Baseline stack frame layout.

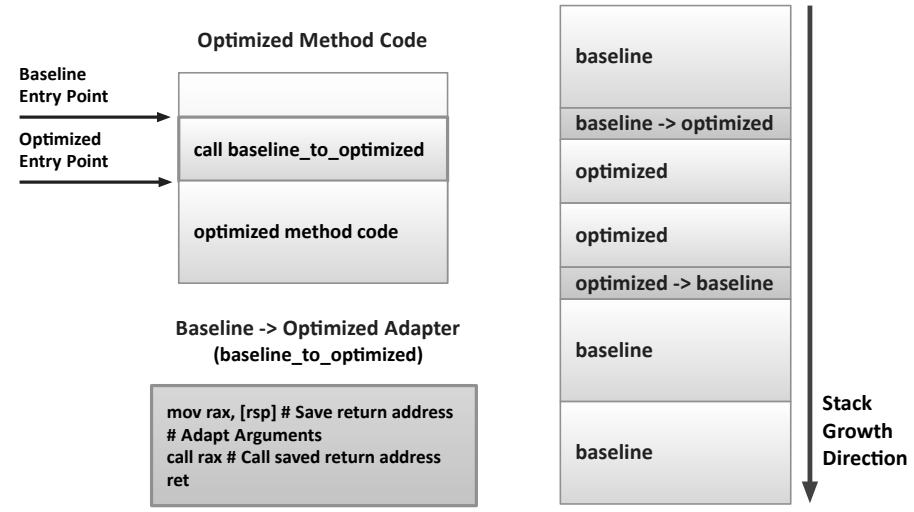
(b) Optimized stack frame layout.

Figure 5.6: Stack frame layouts for Maxine's baseline (a) and optimizing (b) compilers.

method is called from a baseline method, the baseline method calls the optimized method's baseline entry point, and vice versa. In the example in figure 5.7a on the next page the baseline entry point invokes the Baseline → Optimized adapter which first saves the return address in `RAX` and then adapts the arguments. The adapter code then calls the saved return address, which matches the address of the optimized entry point.

Adapter frames are used to interleave baseline and optimized frames on the same execution stack. As the number of methods that are re-compiled using the optimizing compiler increases, the number of adapter frames appearing on the stack also increases. Gradually, as the majority of the baseline methods are re-compiled, fewer frame layout transitions are necessary, thus the number of adapter frames appearing on the stack decreases. This exposes a peculiarity where execution performance decreases initially, as methods are re-compiled with the optimizing compiler (figure 5.8b on the following page). If the overhead introduced by adapter frames is not recuperated by the optimizing compiler, performance may decrease.

This is important to keep in mind since the trace compiler introduces yet another adapter frame, the Baseline → Region adapter. As will be described later, regions are in fact optimized methods that can only be entered from baseline methods; there is no need for an Optimized → Region adapter. Also, a different Region → Baseline adapter is not necessary because it's already handled by the Optimized → Baseline adapter and Region → Optimized adapter is not needed. Figure 5.8d on the next page depicts a scenario where all of the baseline methods in figure 5.8b on the following page make use of trace regions. This exposes a very important observation: the introduction of trace regions increases



(a) Code of an optimized method showing its two entry points and adapter code.

(b) Sample Maxine execution stack with two adapter frames switching between baseline and optimized methods.

Figure 5.7: Adapter frames.

the number of adapter frames on the stack which may in turn lead to a degradation in performance.

(a) Ten consecutive baseline frames.

(b) Five baseline and five optimized frames interleaved with seven adapter frames.

(c) Ten consecutive optimized frames.

(d) Five baseline, five trace regions and five optimized frames interleaved with nine adapter frames.

Figure 5.8: Execution stack with baseline, trace regions and optimized methods interleaved.

5.2.5 Garbage Collection & Safepointing

Maxine uses a stop-the-world, exact and copying garbage collector. Since garbage collection can occur at any point, Maxine must be able exactly identify which locations on the stack

and which machine registers contain references and which do not. Since the copying collector can move objects in the heap, references to these objects must be updated as well. The bookkeeping associated with being able to identify exactly which stack locations and which registers contain references at any point in time is significant. Maxine uses stack and register reference maps that are generated by the compiler to encode this information at select program locations called stop points. Stop points are locations in machine code where the VM can be temporarily suspended and where its state can be inspected and modified. Stop points can appear in several places:

- *Safepoints*: A stop-the-world collector must pause all running threads (mutators) before collecting garbage. Stopping all running threads can be performed several ways: signaling the operating system, or polling. Stopping a running thread using a system call offers no guarantees in terms of where execution is suspended. This is problematic because the collector may have trouble figuring out which registers or stack locations contain references. Some registers may even be involved in computing a reference and there is no easy way to distinguish which registers contain references. Stopping a running thread using polling is safer because the thread can be stopped at predetermined safe points, where the locations of all references are known. Polling usually works by introducing a benign machine instruction in the mutator's code stream that has no side effects unless an external event occurs. Once the external event is triggered, the benign instruction causes a system trap, effectively suspending the thread. Maxine uses the following such instruction: `mov r14, [r14]`. As long as the value at the address stored in register `r14` is the same as the address itself, the instruction has no effect. If the value is set to `zero`, the instruction causes a page fault on a subsequent execution of the instruction, after `r14` becomes `zero`. These are called safepoint polling instructions and are usually introduced at backward branch targets, or loop headers, so that mutators can respond to a thread stop notification in a bounded amount of time. Without safepoints, it's conceivable that a mutator could be spin waiting and never respond to a thread stop notification.
- *Call Sites*: Aside from safepoints, the execution of a method may appear to be stopped right after a call site, if the callee is suspended at a safepoint.

5.3 Maxpath

Maxpath is a full-featured trace-based compiler that is built on top of the Maxine VM. In contrast with previous research on trace-based compilation in Java [GPF06], Maxpath aims to execute significantly larger and more complex Java applications. Our goal with Maxpath is to understand trace-based compilation in a much more realistic setting, where all features of the Java programming language are supported.

5.3.1 Discovering Hot Program Regions

Bala et al. [BDB00] proposed a very effective, yet simple, way of detecting hot program regions in their Dynamo system. Execution counters are used to count the number of backward branches taken at runtime. If this number exceeds a certain threshold, the target of the branch instruction is considered a frequently executed loop header, and naturally all instructions within the loop are presumed to be hot as well. Certainly this is not always the case, often times cold code, or even warm code appears in hot loop regions.

In order to exclude such code from compilation, dynamic path profiling is used to detect which program paths are more frequently executed than others. Previous work on path profiling by Ball [BL96] investigated ways in which information about path execution frequency can be collected and used in tuning performance. In that work, path profiling was accumulated over complete runs of an application. Therefore, any performance benefits would only apply to subsequent compilations and executions of an application.

Collecting accurate path profiling information in the context of a just-in-time compiler is complicated by the fact that only a small fraction of the application behavior is observed. Precise path profiling information would be worthless to a just-in-time compiler if it were only made available when the application terminated. For path profiling information to be useful, it must be made available early on, when the just-in-time compiler can actually make use of it.

Our Maxpath trace-based compiler performs path profiling by sampling the execution of program paths within hot loop regions. Hot loop regions are detected by instrumenting the execution of the targets of backward branches, much in the same way Dynamo detects hot loop regions. During trace sampling, Maxpath records program traces and assembles them into larger structures called *trace regions*. The growth of trace regions is guided by various heuristics. Once a trace region has grown sufficiently, no further traces are accumulated. The trace region is then optimized and linked in for future execution.

Detecting Loop Headers (Anchors)

Previous work on trace-based compilation in the J^2VM and Hotpath VM [Gal06] used dynamic loop header discovery. Hotpath detected loop headers by monitoring the execution frequency of backward branches using a Java interpreter. Once the execution frequency exceeded a given threshold, the branch target would be considered a hot loop header. Hotpath would then begin recording traces starting at the loop header, and continue recording until the loop header was reached again.

We found that this approach does not scale well. First of all, not all backward branch targets are loop headers. The loop detection method used in Hotpath may incorrectly classify some branch targets as loop headers. Secondly, and most importantly, not having a clear view of the loop structure in a control flow graph may lead to the construction of sub-optimal trace regions.

Consider the behavior of Hotpath in the following example:

```
1 for (int i = 0; i < 10; i++) {
2     for (int j = 0; j < 10; j++) {
3         // inner loop body
4     }
5     // outer loop body
6 }
```

In this example, the inner loop header is discovered first since it is executed more frequently than the outer loop header. Traces that cover the inner loop body are then recorded. As additional traces are accumulated, some may escape into the outer loop scope and connect back to the inner loop header. This would have the negative effect of polluting the recorded trace region with the cold code that appears in the outer loop header, leading to poor code quality.

The approach we are taking in Maxpath is to perform a static loop analysis phase as a preprocessing step during class loading. Loop header information can be collected essentially for free during bytecode verification. This allows us to guide the growth of trace regions so that they do not escape block scopes. An additional reason for this design choice is that unlike other interpreter-based trace compilers, Maxpath relies on code instrumentation for profiling and trace recording. Instrumenting branch instructions, rather than branch targets, is more difficult and requires more instrumentation code since branch instructions are more frequent than branch targets. Although loop headers are discovered statically during bytecode verification, we still perform profiling to detect "hot" loop headers.

5.3.2 Trace Recording

The main prerequisite for building a trace-based compiler is the capability to record program execution traces. Building a trace recorder in an interpreter-based execution environment is fairly easy.

Trace Recording using Interpreters

To record traces, recording code can be inserted in the interpreter loop before or after each Java bytecode instruction handler. This code can record the execution of each bytecode as well as inspect results produced by the execution of the instruction. Unfortunately, this code severely impacts the performance of the interpreter's dispatch loop. Since trace recording is a relatively infrequent process, a two-interpreter trace recording system can be used to minimize any trace recording overhead (see figure 5.9a on page 56). One runtime interpreter is used to execute Java bytecode instructions while a functionally equivalent recording interpreter is used to record traces. Once the runtime interpreter decides to record a trace, it can switch to the recording interpreter. Similarly, once trace recording is complete, the recording interpreter can switch back to the non-instrumented interpreter.

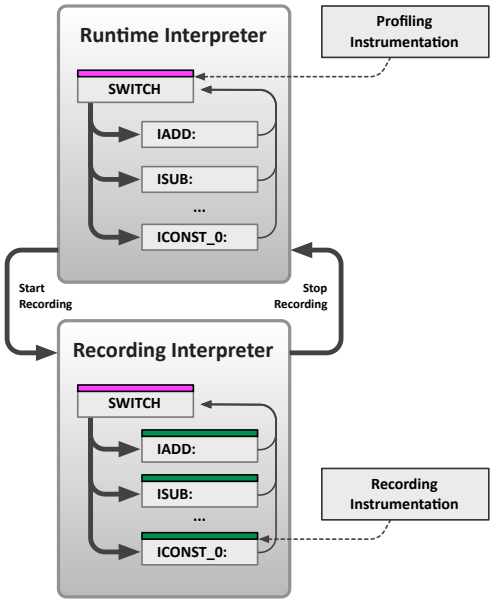
Trace Recording in baseline Just-in-Time Compilers

Since the performance of the baseline just-in-time compiler is much higher than that of a traditional interpreter, it is more difficult to pay back the overhead incurred by trace recording instrumentation. The more efficient the baseline execution environment is, the more important it becomes to reduce instrumentation overhead. In order to reduce the overhead of tracing we use a technique similar to the one used in the dual interpreter approach. Maxpath maintains two compiled versions for each method that is subject to trace recording, a *profile instrumented method* (PIM) version and a *trace instrumented method* (TIM) version (see figure 5.9b on the following page). The profile instrumented version embeds profiling instrumentation or *anchors* at select program locations. These locations are loop headers and are discovered using an inexpensive static analysis performed during Java bytecode verification. During program execution these anchors profile program behavior and trigger the recording of program traces and the compilation of trace regions. Trace recording is performed by hot swapping the profile instrumented version of the method with the trace instrumented version. The trace instrumented version contains tracing instrumentation at each Java bytecode location. This is used to signal the execution of each Java bytecode as it is executed, so that the trace compiler can record every executed instruction. Once trace recording is complete, execution is resumed in the profile instrumented version of the method. Switching between the two versions is possible because they share the same stack frame layout, namely the Java bytecode stack frame layout.

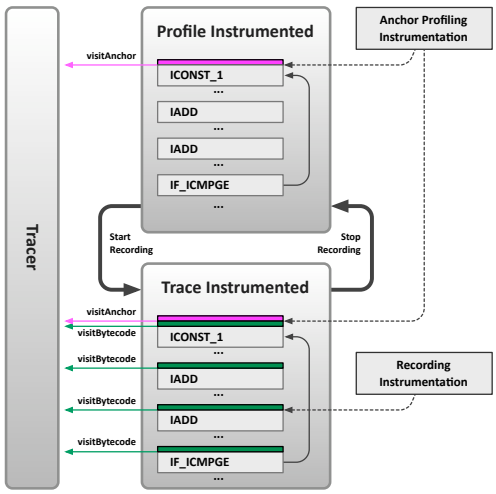
Tracer

The *tracer* is responsible for the interaction between profile and trace instrumented methods, as well as controlling the growth and selection of trace regions. The tracer is a thread-local runtime component that receives messages from instrumented methods and triggers the recording and the execution of trace regions. Instrumented methods interact with the tracer by sending messages such as: `visitAnchor` and `visitBytecode`. The tracer responds with a *resumption address*. This address indicates where the instrumented code should resume execution. If the resumption address is `zero` then execution falls through to the next Java bytecode instruction. Figure 5.11 on page 57 illustrates the implementation of the `visitAnchor` instrumentation code in Java as well as in machine code. This instrumentation code is inserted at each anchor location. In profile instrumented methods, the `visitAnchor` message tells the tracer which anchor is about to execute and what the current frame pointer is. The tracer uses this information to profile the anchor's execution behavior and trigger trace recording. If the tracer wants to start recording, it replies to the `visitAnchor` message with the program address of the bytecode to be recorded in the trace instrumented version of the method. Effectively, execution is transferred from the profile instrumented version of the method to the trace instrumented version (figure 5.10 on page 57). If the tracer wants to keep profiling the execution of the anchor, it replies with the `zero` address.

Tracing Bytecode Execution Once execution is transferred to the trace instrumented method, the `visitBytecode` (figure 5.12 on page 57) message tells the tracer which bytecode is about to be executed. It does this by passing along the current instruction pointer, as



(a) Design of a simple interpreter based trace recorder. To minimize trace recording overhead, two interpreters are used. A runtime interpreter is used to profile and execute instructions, while a recording interpreter is used to record instructions. The VM can switch between the two interpreters at any time.



(b) Design of a baseline just-in-time based trace recorder. To minimize instrumentation overhead, two versions of a method are compiled. The first, profile instrumented, version contains *anchors* that monitor the execution frequency of select program locations and trigger trace recording and compilation. The second, trace instrumented, version contains recording instrumentation that is used to record traces. The two method versions can be used interchangeably because they share a common stack frame layout.

Figure 5.9: Interpreter vs. just-in-time based trace recorder.

well as the stack and the frame pointer. The tracer can infer the executed bytecode from the instruction pointer by using metadata produced during baseline compilation. (Using the stack pointer, the tracer can inspect the current values on top of the Java stack. This is necessary for recording branch conditions or speculating on the receiver types of virtual calls.) After processing the message, if the tracer wants to continue recording, it replies to the message with the `zero` address. Otherwise, if the tracer wants to stop recording it can reply with the program address of the equivalent current bytecode in the profile instrumented version of the method. Effectively, switching back to the more efficient profile instrumented version of the method.

This technique allows Maxpath to record traces with minimal runtime overhead, at the expense of maintaining duplicate method versions (profile and trace instrumented). However, since trace instrumented methods are only ever needed if trace recording actually occurs, they are created on demand and freed at will. Moreover, any residual anchor profiling instrumentation overhead can be minimized through code patching. In Maxpath, we simply overwrite the anchor instrumentation with a `JUMP` instruction that effectively jumps over the instrumentation if it's no longer needed.

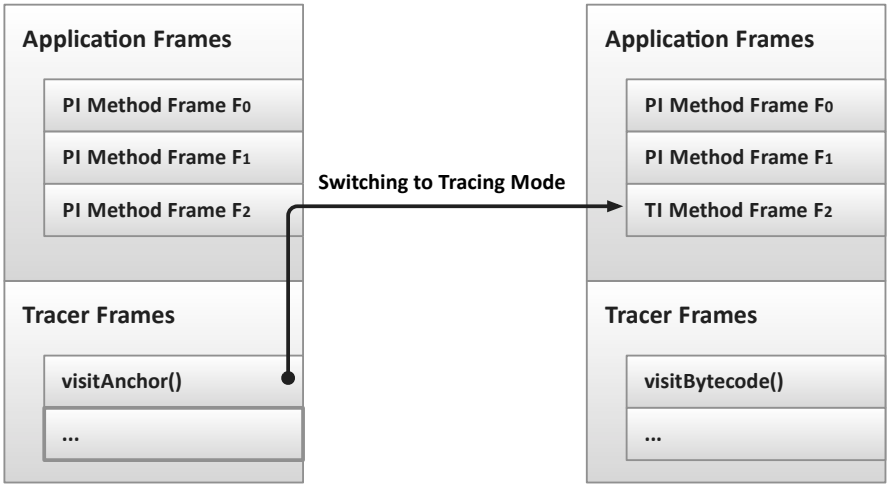


Figure 5.10: Switching between a profiling instrumented method (`PI Method Frame` F_2) to a trace instrumented method (`TI Method Frame` F_2) as a result of a `visitAnchor` message.

```
1 resumeAddr = visitAnchor(anchor, RBP);
2 if (resumeAddr != 0) {
3     jump(resumeAddr);
4 }
```

(a) Java implementation of the `visitAnchor` instrumentation code.

```
1 mov   rdi, #anchor
2 mov   rsi, rbp
3 call  visitAnchor
4 cmp   rax, 0
5 jnz   rax
```

(b) Actual machine code implementation of the `visitAnchor` instrumentation code that is inlined in profile instrumented code.

Figure 5.11: Implementation of `visitAnchor` instrumentation in Java and its corresponding machine code.

```
1 resumeAddr = visitBytecode(RIP, RSP, RBP);
2 if (resumeAddr != 0) {
3     jump(resumeAddr);
4 }
```

Figure 5.12: Implementation of `visitBytecode` instrumentation in Java.

Tracing Method Invocations Method invocations present a challenge to trace recording. Unlike state of the art trace compilers [GES+09], Maxpath is not limited to tracing indefinitely into callees. Doing this would restrict trace formation to leaf methods, which would be impractical. Instead, Maxpath uses an inlining policy that dictates which methods should be partially inlined (or traced through).

Maxpath uses the following inlining policy:

1. Methods that contain loop headers are not inlined, for they may cause the tracer to get stuck unrolling a loop and abort trace recording.

2. If the callee has not been compiled yet, it is compiled with trace instrumentation and is partially inlined.

3. If the method exists but has only been compiled with the optimizing compiler, it is re-compiled using the baseline compiler with trace instrumentation and is inlined. This is done in order to partially inline methods that have been selected as part of the VM boot image (see section 5.2.2 on page 46 for a detailed discussion). For instance, the constructor of the root object class in Java (`Object.ctor()`) is compiled as part of the VM image, not being able to trace through and inline its execution would be a severe limitation.

4. The overall size of the callee method is not part of our inlining policy simply because it does not necessarily reflect the size of the partially inlined would be trace.

Should the method be inlined, its TIM version is invoked allowing the trace recorder to descend into the callee. The invoked method will continue to send `visitBytecode` messages. If the method is not inlined, the tracer is placed on hold and is only resumed once the callee method returns. While the tracer is on hold, waiting for the callee to return, the callee method, or a method further down the execution stack may trigger yet other traces to be recorded and executed. Putting the tracer on hold, may prohibit these traces from being recorded. For this reason, we use a stack of tracer scopes. Once one tracer scope is on hold, waiting for the callee method to return, it is pushed on a tracer scope stack, and a new tracer scope is created that is ready to record and execute additional traces. Once the callee method returns [1] the old tracer scope is popped off the tracer scope stack and recording is continued. At any one point, any number of tracer scopes can be on the stack for any given thread, but only one trace scope is active per thread, while the remaining ones are on hold.

Figure 5.13 on the next page illustrates a scenario where three tracer scopes T_0, T_1 and T_2 are recording traces at various levels on the execution stack, at method frames F_1, F_4, and F_7. The first tracer scope T_0 is anchored at frame F_1 and is on hold waiting for frame F_2 to return. The second tracer scope T_1 is anchored at frame F_3 and is inlining frame F_4 but is also on hold, waiting for frame F_5. The third tracer scope T_3 is the only active tracer scope and tracing the execution of the method at frame F_7.

A similar approach to our explicit tracer scope stack was later used in the SPUR [BBF+10] project. In the SPUR system, tracer scopes are stacked implicitly on the execution stack; each activation record holds a reference to its tracer scope and passes it to the callee.

Threading presents yet another problem. What if two threads are executing the same exact piece of code, and therefore competing for growing the same trace region? In order to avoid lock contention on a shared trace region data structure, we instead grow multiple regions in parallel. The tracers commit the recorded trace regions in completing order, and only the last one to be committed is kept, the rest are discarded.

[1] Detecting that the callee has returned can be done by observing a `visit` message belonging to the caller frame while the tracer is on hold.

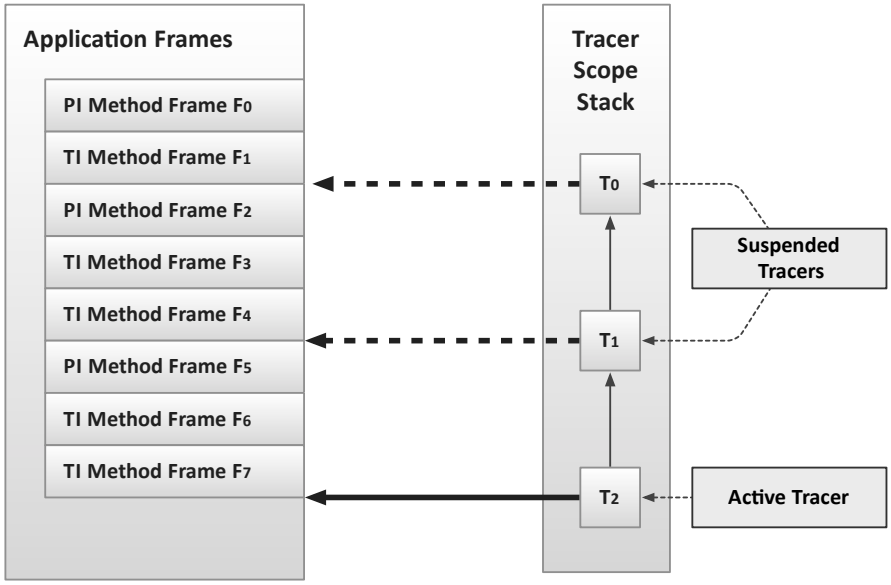


Figure 5.13: Three tracer scopes recording traces at various levels on the execution stack. Tracer scopes T_0 and T_1 are on hold waiting for callee frames to return. Tracer scope T_2 is active, and recording traces.

Trace Aborting Tracing can be aborted for many reasons. The most common reason is when the recorded trace exits the recording scope, leaving the loop, or the method containing the original trace anchor. Another common scenario is when the trace length exceeds a certain limit. In these cases, trace recording can abort gracefully, since the tracer knows exactly where to resume execution. The topmost tracer is popped off the tracer stack, and the new tracer that is now on top of the tracer stack is resumed. A more problematic case is when an exception is thrown as a result of an invoked method. In this case, it would be difficult to track how the VM performs stack unwinding and where execution is ultimately resumed. In this case, we abort all active tracers belonging to the thread throwing the exception, by clearing the tracer stack.

5.3.3 Heuristics

This section describes various heuristics used in trace region formation and other implementation considerations.

Trace Region Formation

The main goal of our region formation algorithm is to:

> *construct regions that cover an application's most frequently executed code paths, exclude most of its infrequently executed code and at the same time minimize*

the number of trace region transitions

By profiling loop headers we can detect frequently executed loops, however we don't necessarily detect the most frequently executed paths through these loops. We assume that the first recorded path through a loop, after the loop has executed a sufficiently high number of iterations, is likely to be the most frequently executed path. The general assumption most trace compilers make is that at runtime, execution will stay on this recorded path and therefore benefit from compiler optimizations. However, this is rarely the case. In practice, aside from a few scientific applications, very few programs have loops that exhibit such predictable behavior. Consider the example in figure 5.14: The two paths through the loop are equally hot. If the first trace is recorded when $i = 100$ then the path of code executed when $i \ \% \ 2 \neq 0$ is not included in the compiled trace region. Half of the remaining 900 iterations of the loop are executed in trace compiled code, while the other half are executed in the baseline compiler. A naive trace execution analysis would conclude that the program's execution speed would increase by $\approx 50\%$, if the trace compiler is known to be twice as good as the baseline compiler. In reality the program's performance would actually deteriorate, for the following two reasons: 1) code cache locality would be impacted because of constant switching between the baseline code and the trace code, and 2) the cost of transferring into and out of compiled traces is relatively high, an overhead which is incurred 450 times.

```
1 1    for (int i = 0; i < 1000; i++) {
2 1        if (i % 2 == 0) {
3 1            sum += 1;
4 1        } else {
5  2           sum += 2;
6 12      }
7 12  }
```

Figure 5.14: A simple loop with two equally hot paths.

In general, the performance gain achieved after compilation for one execution of a compiled region can be expressed as:

$$\begin{aligned} \text{Region Performance Gain} &= \text{Baseline Execution Time} \\ &- \text{Optimized Execution Time} - \text{Transfer Cost} \end{aligned}$$

For trace compilation to pay off, the following must hold:

$$n \times \text{Region Performance Gain} > \text{Profiling Cost} + \text{Compilation Cost}$$

where n is the number of total region executions. If n is small then the compilation and profiling overhead is never paid off. This implies that regions must be formed and compiled early enough so that there is enough benefit to be gained from executing them more efficiently in the long run. However, forming regions early on is also problematic because it leads to the formation of many small regions which are not executed as frequently in the

long run. Another extremely important factor is the region performance gain. If this number is low, then trace compilation does not pay off. In order to increase region performance gain we can do three things:

- Increase the relative difference between the baseline compiler and the optimizing compiler by investing effort in the optimizing compiler. Execution environments that see a huge performance increase from trace compilation generally have a baseline execution environment (usually interpretation) that is orders of magnitude slower than the optimized execution environment (usually just-in-time compilation). Such a large difference between the two execution environments increases region performance gain and ultimately makes trace compilation pay off. (If the difference is small, as is the case in Maxine, making trace compilation pay off is much more difficult.)
- Reduce the cost of transferring in and out of trace regions. Although we have invested a lot of effort in making trace region transfers efficient (section 5.3.5 on page 68), transfer overhead still plays a significant role (as calling conventions still do).
- Reduce the number of transfers by constructing larger regions. This has the added benefit that it increases the region performance gain since more code is available for optimization. Naturally, the value of n decreases since there are fewer smaller regions to execute, but is outweighed by the aforementioned performance gain.

To demonstrate the effects of n and region performance gain in Maxpath we measure the execution performance of the program in figure 5.15a on the following page. The experiment is setup such that `outer` $*$ `inner` $\approx$ `100,000,000`. Maxpath is configured to record only one trace, thus compiling only the inner loop. The value of `inner` is varied from 1 to 32 (plotted on the x axis) and value of `outer` is adjusted accordingly. Figure 5.15b on the next page and figure 5.15c on the following page plots the performance of Baseline and Maxpath in milliseconds for every execution of the program in figure 5.15a on the next page. In both cases, the program reaches peak performance when `inner` is $\approx$ `10` which is intuitive since more work is done setting up the inner loop when the value of `inner` is small.

To cancel out this effect we compare the two plots in figure 5.15b on the following page and figure 5.15c on the next page. Figure 5.15d on the following page plots the speedup of Maxpath over the Baseline. When `inner` $=$ `1` the performance of Maxpath is equal to the Baseline. This is as a result of very little work being done in the compiled trace region, and the transfer overhead is incurred `100,000,000` times. As `inner` gets larger, peak performance levels off at $\approx$ 6x.

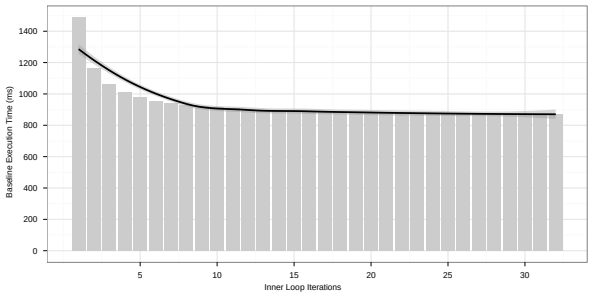
5.3.4 Trace Sampling

Our general trace region formation strategy is to profile the execution of applications in two distinct phases (shown in figure 5.16 on page 63): coarse and fine grained profiling.

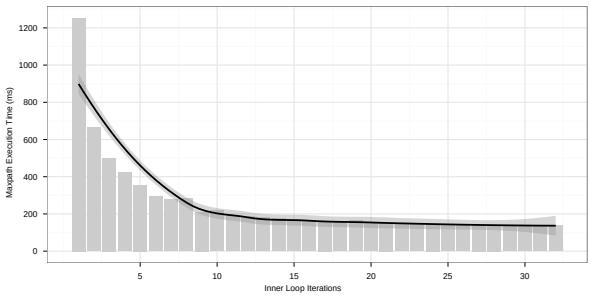
During coarse grained profiling the application is lightly instrumented in order to detect frequently executed program regions. These regions are usually loops, but may also be

```
1 for (int i = 0; i < outer; i++) {
2     for (int j = 0; j < inner; j++) {
3         sum += i + j;
4     }
5 }
```

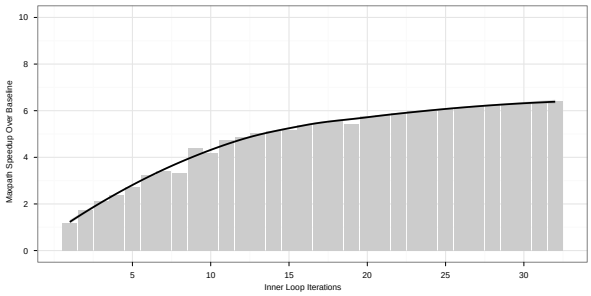
(a) Nested Loop Example



(b) Baseline Performance



(c) Maxpath Performance



(d) Maxpath Speedup Over Baseline

Figure 5.15: Relationship between Baseline and Maxpath in terms of performance.

recursive or leaf methods. Our trace compiler is in no way limited to tracing the execution of loops. In fact, Maxpath profiles and records traces starting at *anchors* which can be sprinkled at arbitrary program locations. However, in practice we have observed that concentrating on loop regions is more beneficial. The primary reason for this is because Maxine's baseline execution environment is relatively close to its optimized execution environment in terms of performance and we have to focus on the program regions that pay off the most.

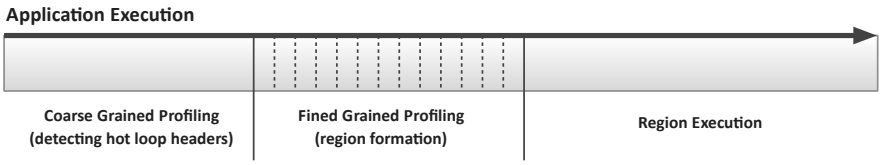


Figure 5.16: Application execution life cycle.

Figure 5.17: Trace region formation and execution life cycle.

Coarse grained profiling is further subdivided into two phases:

1. We have observed that $\approx 80\%$ of anchors in our benchmarks set fail to grow trace regions. Sending `visitAnchor` messages from these anchors is wasteful and negatively impacts performance. To solve this problem we emit instrumentation before each `visitAnchor` that decrements a per-anchor counter variable. Once this counter reaches `zero`, the `visitAnchor` message is sent and the tracer can perform additional profiling. This instrumentation can be encoded efficiently using two x86 instructions: a `sub` followed by a `jz` and it effectively eliminates the overhead of superfluous anchors in PIM code. In TIM code, this type of instrumentation is not used because the tracer needs to be able to observe anchors when tracing program execution, moreover it would not make much of a difference, since programs rarely execute TIM code.

2. The second profiling phase involves sending the `visitAnchor` message and follows the decision diagram shown in figure 5.18 on the following page.

During fine grained profiling we sample several execution traces and incrementally construct trace regions. We call each incremental construction of the trace region an *extension*. Figure 5.17 details the trace formation life cycle.

Processing Tracing Messages

Instrumented code interacts with the tracer using the messages listed in table 5.1 on page 67.

The tracer, in turn, responds with one of three different replies:

- `Continue`: The tracer instructs the sender to continue to the next bytecode instruction.

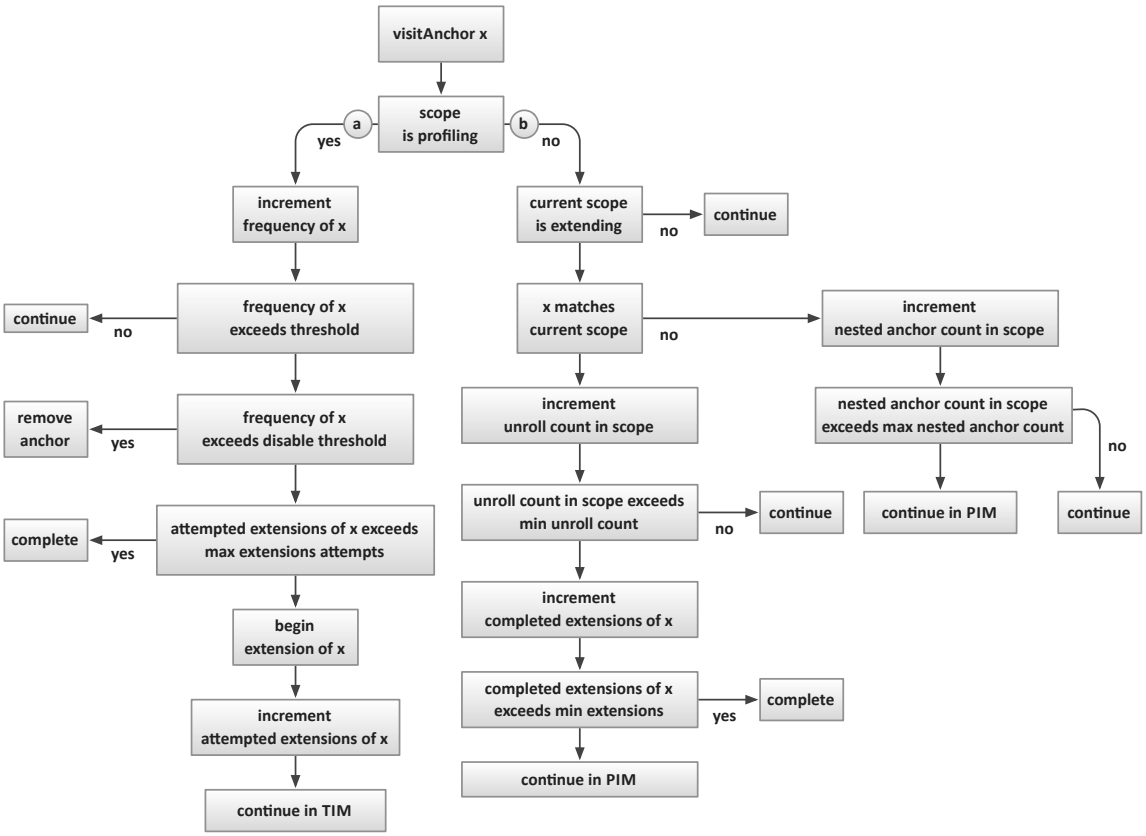


Figure 5.18: Decision diagram for the `visitAnchor` message.

- `Resume in TIM`: The tracer instructs the sender to jump to the current bytecode address in trace instrumented code, effectively swapping the execution of the current frame from a PIM to a TIM.
- `Resume in PIM`: The tracer instructs the sender to jump to the current bytecode address in profile instrumented code. This is usually done when trace recording is being aborted.

`visitAnchor` The decision diagram in figure 5.18 details the inner workings of our trace recorder. Upon receiving a `visitAnchor` message, if the tracer is not currently recording a trace (a), then we increment the execution frequency of the anchor. If the frequency does not exceed a specified threshold then we continue execution in PIM code. If the frequency exceeds a specified disable-threshold then we remove the anchor from PIM code through code patching. This is necessary for cases where an anchor is frequently executed but for some reason we are not able to trace its execution. Attempting to do so, over and over, would lead to poor performance. In essence, we use a threshold window in which we attempt to grow a region, if we fail to complete the trace region within this window, we give up by removing the anchor.

Next we keep track of the number of attempted extensions, this is useful to cap the number

of extensions that have been attempted but have never been completed due to a trace abortion. If we observe too many attempted extensions we complete and link the currently recorded region, otherwise we begin a new extension and resume execution in TIM code. Here, we are swapping the execution of the PIM code that sent the `visitAnchor` message, to TIM code at exactly the same code location. Therefore, the next message received by the tracer is again a `visitAnchor`, but this time it's coming from TIM code, and the current tracing scope is no longer in profiling mode (b), but in trace recording mode.

When a new extension is started a new tracer scope is created and pushed on the tracer scope stack. This new scope keeps track of the anchor from which it was started. If we are in trace recording mode and receive a `visitAnchor`, we first check to see if this anchor matches the anchor associated with the current scope. If they match, then this means we have arrived at the same code location as the one we started recording the trace from, thus completing one path through the loop. Unfortunately, there are several caveats:

- We may have arrived at the same exact code location but not in the same frame. This leads to false loop detection, a situation which was explored by Hayashizaki et al. [HWI+11]. This can happen because of recursive invocation. In order to correctly identify if we've indeed arrived at the same program location, we remember the PIM's frame pointer at the time the tracer scope was created and also check against this. This is the reason for passing the frame pointer (`RBP`) in the `visitAnchor` instrumentation code (figure 5.11 on page 57).
- Another possibility is that we start recording a trace at an anchor a_f in a method f which is invoked twice by another method g. In this case, a_f is actually observed twice, and both times the frame pointer is the same, because each time the frame of f happens to be at the same stack depth. We solve this problem by keeping track of whenever a method is entered and left during trace recording. This is the purpose for the `visitEnter` and `visitLeave` messages. If we detect that we have exited the method from which we have started to record a trace, we simply abort the trace. Intuitively, this is the right thing to do, although it's technically possible to record traces exiting the anchor method. [2] If we were to record traces outside of the scope of anchor methods, we would pollute trace regions with code at call sites, which would be senseless.
- Yet another possibility is that a runtime exception is thrown during trace recording. In this case, the VM unwinds the stack until it finds an appropriate exception handler. Thus, if we start recording a trace at anchor a_f in a method f and an exception occurs that causes the stack to be unwound up to a method g, then it's possible that g may invoke f again leading to a_f being observed twice. The general approach we take with exceptions is to ignore them and abort all trace recording. We've modified Maxine so that it sends all tracers a `visitException` message whenever an exception is thrown or a trap is handled.

If the visited anchor matches the intended current scope then we increment loop unroll count for the current scope. This allows us to unroll several iterations of the loop by simply

[2] This can be done by inserting a guard instruction checking the call site in the caller.

recording its execution several times. Partial loop unrolling optimizations that are found in many sophisticated production compilers can be implemented trivially in a trace compiler.

If we have met a minimum unroll count threshold then we've completed a trace region extension. If we have completed enough extensions then we complete the trace region, compile it, and link it in, otherwise we resume execution in profile instrumented code and wait for more extensions to complete. Attempting to start a new extension right after an extension is completed is generally a bad idea, because it's likely that the new recorded path will not deviate from the previously recorded path. Instead, we use a *backoff* threshold that allows us to ignore several iterations of the loop before attempting to extend the region, thus sampling program execution at regular intervals. [3]

If the visited anchor does not match the current scope, then this could be for several reasons:

- One possibility is that the current scope anchor a_s is a loop header and the visited anchor a_v is a nested loop header. In this case we are receiving as many a_v's as there are inner loop iterations. We bound the number of times we unroll the execution of an inner loop by keeping track of the number of times we've observed a nested loop header. If this number exceeds a certain threshold, we abort trace recording.
- Another possibility is that a_s is a nested loop header and a_v is the outer loop header. This can occur if we're unrolling the execution of the outer loop by tracing the last iteration in the inner loop, which leads back to the loop header of the outer loop before its own. Tracing the outer loop is a good idea, because it allows us to reduce the number of trace region transfers and capture more contextual control flow which is useful for optimizations.

`visitBlock` & `visitBytecode` Trace instrumented code sends `visitBlock` and `visitBytecode` messages before each basic block and bytecode instruction. Maxpath uses these notifications to construct a control flow graph (CFG) and to perform additional profiling. For example, `visitBlock` messages can be used to count the execution frequency of basic blocks and compute path profiles, or even inspect the runtime execution state at basic block entry points. Although Maxpath could rely only on coarse grained messages such as `visitBlock`s for CFG construction, we opted for more fine grained control because it makes trace recording and partial inlining easier. A precise view of how bytecodes are executed up to, through, and after a partially inlined callee allows us to tail duplicate the remaining portion of the callee's basic block (after the call site) and attach it to the partially inlined path before the return block in the callee. A coarse, block level, view would not allow us to observe the execution after the callee returns and miss out on the remaining portion of the callee's basic block.

Frame Unwinding During trace recording, if we are inlining several methods deep, our stack appears as a sequence of TIM methods such as: $\{\ldots tim_0, tim_1, \ldots, tim_n\}$. If we abort

[3] Experimentally, we have found that using a prime number or a randomly generated number as a *backoff* is best because it tends to expose more execution paths.

Message Type	Description
`visitAnchor`	Sent from both PIM and TIM code. These messages are sent as long as the instrumentation code sending them has not been removed through code patching.
`visitBlock`	Only sent from TIM code before a basic block is executed and before any `visitAnchor` or `visitBytecode` messages are sent.
`visitBytecode`	Only sent from TIM code before a bytecode is executed and after any `visitAnchor` messages are sent.
`visitInlineMethod`	Only sent from TIM code before a callee is invoked and after any `visitBytecode` messages are sent. These messages are used to by the TIM code at call sites in order to select PIM or TIM callees.
`visitEnter`	Only sent from TIM code before the first basic block in a method is executed.
`visitLeave`	Only sent from TIM code after the a return instruction is executed.
`visitException`	Sent from the VM whenever an exception is handled, this can occur at any time.

Table 5.1: Message Types: Instrumented code interacts with the tracer by sending 7 different types of messages.

trace recording, we have to replace all these frames with PIM methods, thus we would like to see: $\{\ldots pim_0, pim_1, \ldots, pim_n\}$. This can be done by patching all return addresses on the stack to point to PIM method instead of TIM methods but this is unnecessarily complicated. Instead, we replace the top most frame with a PIM method ($\{\ldots tim_0, tim_1, \ldots, pim_n\}$) and resume execution in PIM code. When the top most pim_n frame returns into tim_{n-1} execution will be resumed in TIM code. Execution in TIM code will be inconsistent with what the tracer expects. Whenever a message from TIM code is sent to the tracer, the tracer then unwinds the top most frame again until all frames are unwound.

Reentrancy Because of the meta-circular nature of Maxine, it is very likely that the execution of the tracer itself would trigger tracing as well. The tracer would end up tracing itself. We cannot simply disable the insertion of anchors in any code that is reachable from the trace compiler because we would exclude large portions of the JDK. Instead, we prevent reentrant execution into the tracer using a lock that we toggle every time we enter and exit a tracing routine. If a message is received by the tracer while the lock is toggled, then it's simply ignored.

Garbage Collection Constraints If a garbage collection were to occur during tracing, Maxine would need to walk the stacks of all running threads. This means we can never deviate from the way Maxine handles stack frames, and we must always ensure that stack reference maps are available at all call sites (see discussion on various stop positions in section 5.2.5 on page 51). To enforce this invariant, we always make sure to encode stack reference maps at all call sites in instrumented code, and also at all code locations that can be potentially patched with a call instruction.

5.3.5 Trace Region Compilation & Execution

In order to cooperate with Maxine's runtime infrastructure, we model trace regions as methods. Stack walking and garbage collection rely on the method as being the sole computational element. Modeling trace regions as methods is the most natural way to fit into Maxine's runtime environment. An additional benefit of this approach is that we are able to use Maxine's back-end register allocator and code generator.

One crucial distinction between trace regions and methods is that trace regions have many exit points, while methods have only one. In a method, all returning control-flow can be joined into one exit point. Moreover, unlike methods, trace regions have many live-out parameters, while Java methods have at most one.

Efficient transfer of control to, and out of trace regions, is very important. We have observed that on average, trace regions have approximately 3 times as many live-in variables as methods, (as shown in figure 9.13 on page 127), and this represents only a small subset the total number of local variables that are lived at the program locations were trace regions are anchored. Although we ignore all unused live variables, many values must still flow into trace regions, and therefore an efficient calling convention is necessary.

We have tried various approaches, ranging from passing live variables by value or by reference, or a combination of both. These approaches incur a significant overhead since they require quite a bit of parameter shuffling to occur on the stack.

In the end we took a more efficient approach. We statically link (using a direct method call) the trace region to anchor call sites in methods compiled with the baseline just-in-time compiler. Since we have complete information about stack frame layouts, we can overlay the parameters of trace regions (which are modeled as methods) to the exact stack locations of live-in variables located in baseline method frames. This way, compiled regions have efficient and direct read or write access to live-in variables. In order to do this we modify the register allocator so that it maps input parameters to stack locations in the caller's frame. Live-out variables are handled similarly, they are written back directly into the caller's method frame. This is possible because a trace region is only called from one call site.

Transfers

One of the challenging aspects of trace-based compilation is building an efficient control-transfer mechanism. Although trace regions attempt to capture as much control flow as possible, a significant number of trace region exits still occur, requiring execution to be resumed in baseline methods.

In order to implement transfers, we have introduced a new control-flow instruction to the Maxine VM named `Transfer`. This instruction carries with it enough information to reconstruct a sequence of stack frames, write back appropriate values, and then resume execution at a different program location. This information is essentially the symbolic Java stack state

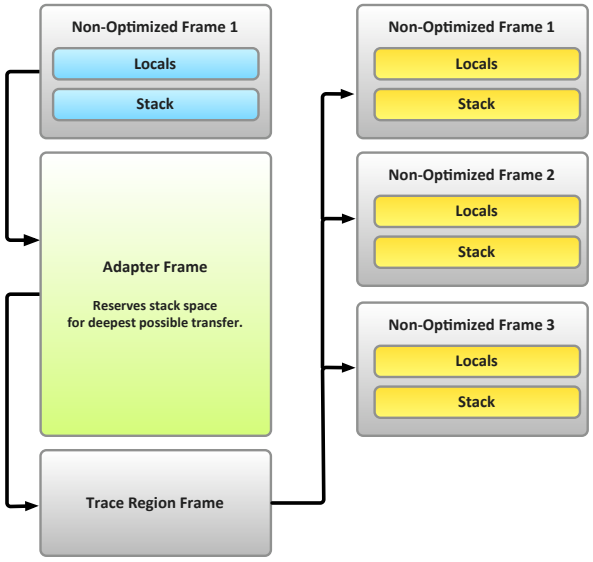


Figure 5.19: Invocation of a trace region from a baseline method. An adapter frame reserves space on the stack for the deepest possible transfer. The trace region frame has direct access to the live-in variables in the original baseline method frame. Upon exiting, the trace region writes live-out variables and reconstructs additional baseline method frames, before resuming execution.

that was recorded during trace recording.

Fast Transfers During code generation, for each transfer instruction, machine code that writes back each individual live-out variable is emitted. The last machine code instruction emitted is an unconditional branch to a bytecode address in PIM code where execution should resume. Because transfers may restore multiple baseline frames, as is the case if a region exit occurs in a deeply inlined method, we need to ensure that enough space is reserved on the stack for the deepest possible region exit. If we don't reserve enough space, the restored methods frames may overwrite the frame of the currently executing trace region. To accomplish this, we use a trace region adapter frame that allocates enough space on the stack before calling the trace region (see figure 5.19).

The code generated for each transfer instruction could lead to an increase in compilation time, and code cache size. For this reason we have implemented two types of fast transfers (show in figure 5.20 on the following page):

- *Inlined Fast Transfer Stubs* are emitted inline with the emitted trace region code. As an optimization, we arrange all transfer stubs at the end of the region method code so that they don't interfere, in terms of code locality, with the compiled traces.

  Unfortunately, early research has show that $\approx$ 80% of the total generated code is made up of transfer stubs, especially if we perform deep inlining, or if we guard against exceptions. Early trace compilers for Java [GPF06] did not support throwing

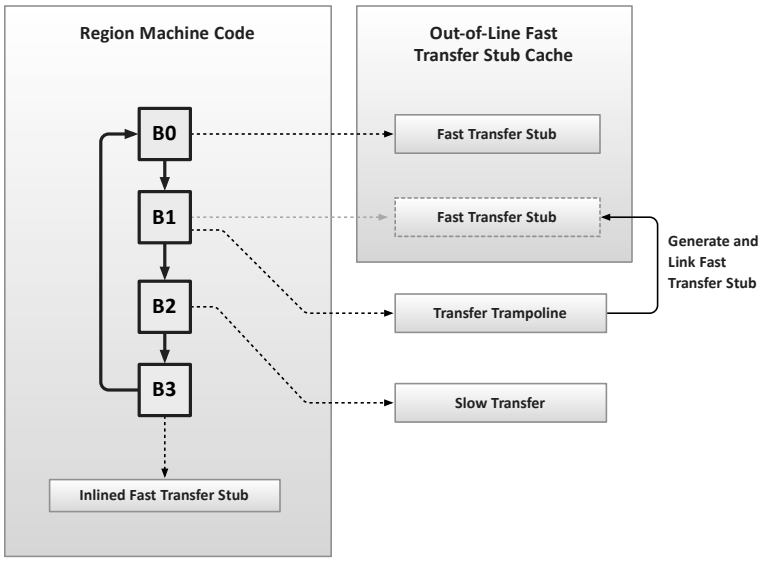


Figure 5.20: Three transfer types: Inlined, Out-of-Line and Slow Transfers.

Java exceptions from within compiled traces. Instead, the exception condition was guarded against, and if it were to fail, control would be transferred to the interpreter, where the exception would be subsequently thrown and handled. For instance, the expression `x / y` required the guard `y != 0`, which was both inefficient and the cause of many unnecessary transfer stubs. In Maxpath, since we model regions as methods, we can reuse the exception handling mechanism that the method compiler uses, and therefore we can eliminate the need to insert guards before all operations that can throw exceptions implicitly.

- *Out-of-Line Fast Transfer Stubs* In our benchmark applications, we have observed that a relatively few number transfers are actually ever executed at runtime. Therefore it would be wasteful generate transfer stubs for them a priori. A better technique would be to lazily construct transfer stubs as they are needed, and this is in fact the approach we have taken in Maxpath.

  Instead of emitting the transfer stub inline with the trace region code, we emit a call to a trampoline routine which lazily constructs and links in the transfer stub. At the trampoline call site, we encode the current target location of each live-out variable (e.g., whether the variable was stored in a register or spilled to a stack location) into an auxiliary data structure. The trampoline routine first saves all registers and then inspects the return address in order to discover the code location it was called from. The trampoline then generates a transfer stub using the previously encoded data and installs it in the Transfer Stub Cache. Next, the trampoline patches the trampoline call site as well as the return address on the stack and with the address of the generated transfer stub. Finally, it restores the all saved registers and returns. By returning, the trampoline effectively transfers control to the transfer stub. All subsequent executions of the transfer will be handled directly by the generated transfer stub.

Slow Transfers If a transfer is known to happen infrequently, we have an additional transfer mechanism that does not involve any runtime code generation. Instead of writing back each value using custom generated machine code, we instead invoke a `transfer()` method within the Maxpath runtime. This method uses a special calling convention where it first saves the state of all machine registers before executing (similar to the way a trap handler would work). The transfer method then inspects metadata about the transfer which indicates the target location of each live-out variable. Using this data, it can reconstruct baseline frames in the reserved stack region and resume execution. This technique is not as efficient as the fast transfer method, but can be used to reduce code size if certain transfers are known to be executed infrequently.

Transfer of Control Example: Bubble Sort

The Java program in figure 5.21a on page 73 is a simple implementation of the Bubble Sort algorithm. A disassembled version of the program is presented in figure 5.21b on page 73. Maxpath detects that the anchor at bytecode position 42 is frequently executed and compiles a region for it. The compiled region, shown in figure 5.22b on page 74, contains only one transfer point, to bytecode position 50.

Maxpath transfers control to the trace region code by patching the anchor instrumentation in the baseline method with a `call` instruction to the beginning of region code. The layout of the baseline method frame for the `bubble(int[])` is shown in figure 5.22a on page 74. The baseline compiler uses the `rbp` register to index local variables and the `rsp` register for the operand stack.

After the call, the region code (figure 5.22b on page 74) decrements the `rsp` pointer to make room for the deepest possible transfer as well as for its own local temporary variables. The region has three live-in variables: `list`, `i` and `j` each of which are currently stored in the caller's frame at `[rbp + 96]`, `[rbp - 32]` and `[rbp - 48]` respectively. The `rbp` register is never modified by the region, and always points into the caller's frame. These original locations for live-in variables are given to the region register allocator, which decides to map `i` → `eax` and `j` → `edx`, and leave `list` in its original location `[rbp + 96]` in the caller frame. Since `list` is a reference, Maxpath must ensure that its location is covered by a reference map; this responsibility is delegated to the caller frame, which as previously mentioned, maintains reference maps for all code positions that may be patched with `call` instructions.

The transfer stub in figure 5.22c on page 74 is generated on demand using the variable-register mappings computed by the register allocator. The stub writes back the current values of `j`, `i`, by indexing the `rbp` register which is still pointing to the original baseline frame. The current value of `list` does not need to be written back, for it is still in its original location (`[rbp + 96]`). Next, it uses the `lea rsp, [rbp - 48]` to restore the value of `rsp` making it point to the bottom of the operand stack. Finally, it jumps to the code address for bytecode position 50 in PIM, and thus completes the execution of a trace region.

5.4 Conclusion

We have shown that trace-based compilation is feasible in meta-circular virtual execution environments without interpreters by use of dynamic instrumentation. More importantly, we have demonstrated that a Java virtual machine can be retrofitted to support trace-based compilation with relatively little effort, a result that is likely to make trace-based compilation attractive to implementers of well established, production Java virtual machines.

The work in this chapter was in part presented at the International Conference on Principles and Practice of Programming in Java, Vienna, 2010.

```java
public static void bubble(int [] list) {
  int temp;
  for (int i = 0; i < list.length; i++) {
    for (int j = 1; j < (list.length - i); j++) {
      if (list[j - 1] > list[j]) {
        temp = list[j - 1];
        list[j - 1] = list[j];
        list[j] = temp;
      }
    }
  }
}
```

(a) Bubble sort algorithm implemented in Java.

```
public static void bubble(int[]);
  Code:
   ...
   0:   iconst_0
   1:   istore_2
   2:   goto    53
   5:   iconst_1
   6:   istore_3
   7:   goto    42
   10:  aload_0
   11:  iload_3
   12:  iconst_1
   13:  isub
   14:  iaload
   15:  aload_0
   16:  iload_3
   17:  iaload
   18:  if_icmple   39      // Conditional Branch
   21:  aload_0
   22:  iload_3
   23:  iconst_1
   24:  isub
   25:  iaload
   26:  istore_1
   27:  aload_0
   28:  iload_3
   29:  iconst_1
   30:  isub
   31:  aload_0
   32:  iload_3
   33:  iaload
   34:  iastore
   35:  aload_0
   36:  iload_3
   37:  iload_1
   38:  iastore
   39:  iinc    3, 1
   42:  iload_3             // Trace Region Anchor
   43:  aload_0
   44:  arraylength
   45:  iload_2
   46:  isub
   47:  if_icmplt   10
   50:  iinc    2, 1       // Trace Region Transfer Point
   53:  iload_2
   54:  aload_0
   55:  arraylength
   56:  if_icmplt   5
   59:  return
   ...
```

(b) Bytecode disassembly of program in figure 5.21a

Figure 5.21: Bubble sort algorithm in Java and JVML Bytecode

```
   rbp + 104    | 27  | local 0 [filler]
   rbp + 96     | 26  | local 0 [parameter 0] <== $list
   rbp + 88     | 25  | return address
   rbp + 80     | 24  | callers rbp
   rbp + 72     | 23  | template slot 9
   ...                | template slot 8 ... 1
=> rbp + 0      | 14  | template slot 0  <== Baseline Frame Pointer (rbp)
   rbp - 8      | 13  | local 1 [filler]
   rbp - 16     | 12  | local 1 [non-parameter 0]
   rbp - 24     | 11  | local 2 [filler]
   rbp - 32     | 10  | local 2 [non-parameter 1]  <== $i
   rbp - 40     | 9   | local 3 [filler]
   rbp - 48     | 8   | local 3 [non-parameter 2]  <== $j, <== Baseline Stack Pointer (rsp)
   rbp - 56     | 7   | operand stack 0 [filler]
   rbp - 64     | 6   | operand stack 0
   rbp - 72     | 5   | operand stack 1 [filler]
   rbp - 80     | 4   | operand stack 1
   rbp - 88     | 3   | operand stack 2 [filler]   // filler slots ensure 16 byte alignment
   rbp - 96     | 2   | operand stack 2
   rbp - 104    | 1   | operand stack 3 [filler]
   rbp - 112    | 0   | operand stack 3
```

(a) Baseline Frame Layout for `bubble(int [])`

```
     subq      rsp, 0x40          ;; Reserve enough stack space region frame and transfer.
     mov       eax, [rbp - 48]    ;; Copy $j ([rbp - 48]) to eax
     mov       edx, [rbp - 32]    ;; Copy $i ([rbp - 32]) to edx
     jmp       L3: +127
L1 : mov       rbx, rax
     subl      ebx, 0x1
     mov       rcx, [rbp + 96]    ;; Copy $list ([rbp + 96]) to rcx
     movsxd    rdi, [rcx + 16]
     cmp       ebx, edi
     jnb       L5: +133
     ...
     mov       rcx, [rbp + 96]    ;; Copy $list ([rbp + 96]) to rcx
     mov       rcx[rax * 4 + 24], esi
L2 : addl      eax, 0x1
L3 : mov       rbx, [rcx + 16]
     sub       ebx, edx
     mov       r14, [r14] {safepoint}
     cmp       eax, ebx
     jl        L1: -144
     call      L4: +0 {transferTrapStub} ;; Call to out-of-line fast transfer stub.
     ;; Transfer Stub Exit State Map:
     ;; at bubble(Bubble.java:36) [bci: 50]
     ;;              |list          |temp       |i      |j
     ;;    locals:   |[rbp + 96]:a  |<illegal>  |rdx:i  |rax:i
L4 : nop
     nop
L5 : mov       [rsp + 8], rcx
     mov       [rsp + 16], rbx
     call      L6: +0 {stub-throwArrayIndexOutOfBoundsException}
```

(b) Compiled Trace Region

```
                                ;; $list already in [rbp + 96], do not restore
     mov       [rbp - 32], rdx  ;; restore $i
     mov       [rbp - 48], rax  ;; restore $j
     lea       rsp, [rbp - 48]  ;; restore rsp
     jmp       0x104a4cd60      ;; resume in PIM at Bytecode Position 50
```

(c) Generated Transfer Stub

Figure 5.22: Transfer of Control

Chapter 6

Trace Driven Incremental SSA Form Construction

Static Single Assignment (SSA) form [CFR+91] has become ubiquitous in compilers as an intermediate program representation. SSA form is a program transformation where every variable is assigned exactly once. Its use simplifies the way we reason about programs and makes many classic optimizations more effective. Unlike other compilers that use SSA form, our trace-based compiler builds SSA form incrementally, as new traces are collected. This is necessary because trace recording and trace region extensions happen interleaved at runtime, i.e., there is no clear and static view of the CFG we are building SSA form from. At any time, a trace can be added to an already existing region. This means we need a mechanism by which an SSA graph can be extended after it has been built. Also, trace recording can be aborted for various reasons. In such cases, we must undo any changes made to the SSA graph. All previously published algorithms to build SSA form (see for example [BP03, DR05, AH00]) rely on the control flow graph being fixed during construction.

As an additional benefit, building SSA form incrementally gives us fine-grained control over tail duplication: We can freely decide whether to join a newly recorded trace with an existing block or to create a new tail-duplicated block.

6.1 State Vectors

At each program point p we define the program's state at that point as s_p, where s_p is the state vector $\langle s_{bci} : l_1, l_2, ..., l_n \mid k_1, k_2, ..., k_m \rangle$. The Java bytecode position s_{bci} (byte code index) followed by a list of n local [1] $(l_1, l_2, ..., l_n)$ and m stack $(k_1, k_2, ..., k_m)$ SSA values. The state vector is an abstract state representation of a Java method frame and does not encode the state of the heap. For simplicity we make no distinction between stack and local values and simply represent the frame state vector as a list of n SSA values $\langle s_{bci} : v_1, v_2, ..., v_n \rangle$.

[1] In Java bytecode, locals may represent either parameters or local variables.

Operation	Function
`LOAD_X`	$\langle s_{bci} : ..., v_x, ... \rangle \Rightarrow \langle s_{bci+1} : ..., v_x, ... \mid v_x \rangle$
`STORE_X`	$\langle s_{bci} : ..., v_x, ... \mid y \rangle \Rightarrow \langle s_{bci+1} : ..., y, ... \rangle$
`ADD`	$\langle s_{bci} : ... \mid a, b \rangle \Rightarrow \langle s_{bci+1} : ... \mid add(a, b) \rangle$
`MUL`	$\langle s_{bci} : ... \mid add(a, b), sub(b, 1) \rangle \Rightarrow \langle s_{bci+1} : ... \mid mul(add(a, b), sub(b, 1)) \rangle$

Figure 6.1: Four examples of Java bytecode that operate on SSA state vectors.

The representation of a program's state needs to be sufficiently flexible to account for inlining and other inter-method analysis and optimizations. For this reason we chain frame state vectors in a singly linked list where each parent frame state vector describes the state of a program at a call site. The chained inlined state $s = \langle s^x \leftarrow ... \leftarrow s^1 \rangle$ is then simply represented by the state vector s of x states as:

$$s = \langle \langle s^x_{bci} : v^x_1, ..., v^x_{n_x} \rangle \leftarrow ... \leftarrow \langle s^1_{bci} : v^1_1, ..., v^1_{n_1} \rangle \rangle$$

6.1.1 SSA Values

An SSA value is the name of a variable (value) definition. It may be a constant, a parameter, or a function of other SSA values. At each program location, a state vector indicates which SSA values are located in which local variables or stack slots. Using abstract interpretation [CC77], SSA state vectors can be mutated to reflect the flow of data through a sequence of bytecode operations. Formally, each bytecode operation has a function of the form $f_{op}(s_p) \Rightarrow (s'_p, v)$ that returns s'_p, a mutated copy of the input program state s_p and the SSA value it produced (v). The listing in figure 6.1 shows three example bytecode operation functions that mutate state vectors: `LOAD_X` pushes the x^{th} local variable slot on the stack, `STORE_X` stores the top of the stack in the x^{th} local variable slot, and `ADD` creates an $add()$ SSA value of the top two SSA values on the stack and pushes it back onto the stack. One aspect worth noting is that stack load/store operations are eliminated during SSA form construction as a result of the implicit copy propagation that occurs during abstract interpretation.

6.2 SSA Control Flow Graph

Abstract interpretation allows us to generate SSA form for linear sequences of program code (or basic blocks). To model arbitrary control flow we must link basic blocks together and merge state vectors at program merge points (figure 6.2 on the following page). Our algorithm uses abstract interpretation to generate the SSA IR (intermediate representation) for each basic block independently. And then, with the help of SSA Value Forwarding (section 6.2.1 on the next page), it joins basic blocks and inserts ϕ (phi) functions to merge

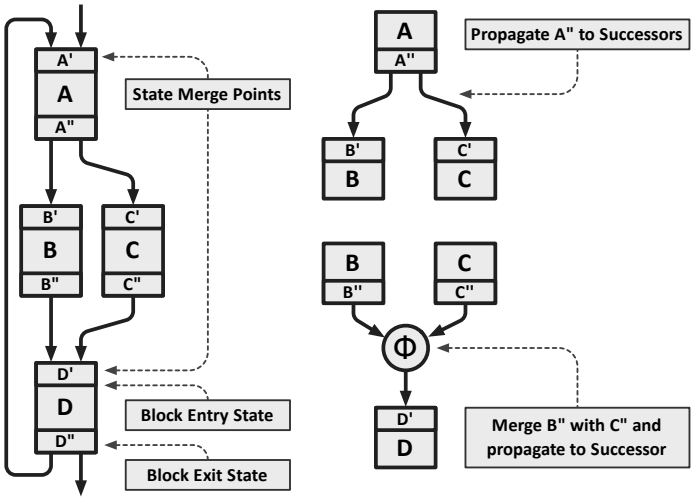


Figure 6.2: Entry and exit state vectors are merged. Each block's exit state is either directly propagated to its successor block, or propagated through a state vector merge function, namely the ϕ function.

control flow at program merge points. Phi functions are of the form:

$$
\begin{aligned}
&s'' = \phi(s, s', ...) \Rightarrow \\
&s'' = \phi(\langle s_{bci} : v_1, v_2, ..., v_n \rangle, \langle s'_{bci} : v'_1, v'_2, ..., v'_n \rangle, ...) \Rightarrow \\
&s'' = \langle s''_{bci} : \phi(v_1, v'_1, ...), \phi(v_2, v'_2, ...), ..., \phi(v_n, v'_n, ...) \rangle
\end{aligned}
$$

The function, $\phi(v_n, v'_n, ...)$ selects one of the operands depending on where control flow arrives to the ϕ function from. It is important to note that ϕ functions are applied to state vectors, and thus, the selection of all individual components must happen simultaneously. Cycles may appear between the operands of ϕ functions due to copy propagation and it is important to consider the ϕ operand selection as if it happens in parallel.

6.2.1 SSA Value Forwarding

Many optimizations and algorithms need to replace one SSA value with another. A naive implementation would require that all uses of an SSA value definition are maintained in an auxiliary def-use chain data structure. Whenever an SSA value is replaced with another, all uses of the original value are updated to refer to the updated value. Unfortunately, maintaining def-use chains can become cumbersome and inefficient, as it requires that all uses be updated whenever an SSA value is replaced with another. Instead, we lazily evaluate each use of an SSA value by following a chain of forwarding pointers. Each SSA value has a forwarding pointer. If this pointer is set, then the SSA value is logically replaced with the value it is being forwarded to. To do this we use the forwarding function $forward(i, j)$ to replace i with j ($i \rightarrow j$). This way, whenever an SSA value is replaced by another, the old SSA value is simply forwarded to the new one by linking the two with a forwarding pointer.

Therefore, the identity of an SSA value at a use site is defined by the value resolution function:

$$resolve(x) = \begin{cases} x & \text{if not forwarded} \\ resolve(x.forward) & \text{otherwise} \end{cases}$$

The chain of forwarding pointers must be followed whenever an instruction is accessed. The current "real" value of an SSA value i is given only by $resolve(i)$.

The resolution function also performs pointer swizzling[2] in order to limit the value forwarding chain to at most one link. If a value i is forwarded to j ($i \rightarrow j$), and j is then forwarded to k ($j \rightarrow k$), then whenever i is resolved it is automatically forwarded to $resolve(i)$. This effectively caches value resolution and is possible for the following reasons:

- An SSA value cannot be forwarded to itself ($i \rightarrow i$). This is to prevent cycles in the forwarding chain. Although the SSA value graph may contain cycles, the forwarding chain is always acyclic.
- If a value i is forwarded to j ($i \rightarrow j$), then i cannot be forwarded again, since i is immediately replaced with j ($i \neq resolve(i)$). Therefore, there is no reason to ever invalidate the value resolution cache.

6.2.2 Basic Block Construction & State Vector Linking

Our incremental SSA form construction technique can be applied to a static control flow graph with relative ease. We first describe the algorithm in the context of a static CFG, and then generalize it for to the dynamic CFG case. The SSA form construction algorithm (algorithm 1 on page 80) operates in three phases:

Phase 1: SSA Value Graph Construction The first phase of the algorithm constructs an SSA value graph for each of the basic blocks in the CFG using abstract interpretation. Each basic block starts out with an initial SSA entry state, which is created by looking at the entry type state of the basic block. The type state is a state vector of typed values and is computed by the bytecode verifier during class loading at each basic block's entry point. The initial SSA entry state of a basic block is filled with SSA parameter values of the type indicated by the entry type state. SSA parameter values are placeholder values representing the incoming values into a basic block. For each bytecode operation the algorithm then constructs SSA values by successively applying each bytecode's operation function. The application of the operation function mutates the state vector and creates a new SSA value which is then added to a sequence of SSA instructions/values in the basic block. The SSA exit state for the basic block is recorded after the application of the last bytecode operation function. This phase of the algorithm constructs an acyclic SSA graph for each basic block rooted in SSA parameter values.

Phase 2: Basic Block Linking & State Forwarding The invariant maintained by the bytecode verifier during class loading is that the declared type states along each control flow

[2]Pointer swizzling is similar to path compression in the Union-Find data structure.

edge between basic blocks are equivalent (or assignable) (denoted by the $\approx$ symbol). This implies that SSA states are also assignable along control flow edges since there is a direct correspondence between type states and SSA states at each program point. More formally, the implication guarantees that $\forall b \in B, \forall b' \in successors(b) : b.exitState \approx b'.entryState$, where B is the set of basic blocks in a CFG. This means that for all blocks, the *exitState* can be propagated to all successor blocks. This is done by forwarding the entry state of each block to its predecessor's exit state (we define state forwarding as: $s \rightarrow s' \Rightarrow forward(s, s') \Rightarrow forward(v_1, v'_1), ..., forward(v_n, v'_n)$).

Each basic block may have one or more predecessor blocks. If a block has only one predecessor, its entry state is forwarded to the exit state of the predecessor block. If a block has two or more predecessors, its entry state is forwarded to a ϕ function merging the exit states of predecessor blocks. At this point, the block's entry state, which used to contain SSA parameter values, is forwarded and swizzled away to the forwarded values. Since the ϕ function is distributed across each component of the state vector, we avoid the insertion of ϕ functions whenever we notice that it is merging equal SSA values, and simply forward to the operand instead. This optimization is sensitive to the order in which we link basic blocks. The pseudocode in algorithm 1 on the next page purposely fails to indicate a block linking order because it is not necessary for correctness. However, since reverse postorder is usually the most efficient block processing order for forward dataflow problems, we can use it to drastically reduce the number of inserted ϕ functions. This is because we can guarantee that a block's predecessor blocks have already been linked and their entry states forwarded, and thus we don't need to worry about inserting ϕ functions for SSA parameter values that may have propagated through predecessor blocks. This phase completes SSA construction.

Phase 3: ϕ Function Elimination The third and last phase of the algorithm simplifies the SSA graph by iteratively eliminating ϕ functions. Functions of the form $v_2 = \phi(v_1, v_1, ..., v_1)$ are forwarded to their common operand ($v_2 \rightarrow v_1$). Functions of the form $v_2 = \phi(v_1, v_2, ..., v_2)$ are forwarded to their incoming operand ($v_2 \rightarrow v_1$); depending on control flow, v_2 can either be equal to itself or to v_1, hence it must be equal to v_1. This form usually appears when phi functions are inserted prematurely for variables that are not updated in a loop body.

This is a fixed-point algorithm which eliminates ϕ functions until no further progress is made.

6.3 Incremental SSA Form Construction

The algorithm described in the previous section is suitable for constructing SSA form in a static compiler because it relies on a fixed CFG and a fixed set of bytecode operations. Our trace compiler, however assembles trace regions by recording multiple execution traces during the execution of the program and linking these together as new program paths are explored. Moreover, one key distinction between static compilers and our trace compiler is that the latter has access to concrete runtime values during trace recording which it can use to generate speculative execution paths within the CFG.

input : A CFG of basic *blocks* that are linked and contain Java bytecode operations but are not in SSA form yet

↪ *Phase 1: Perform abstract interpretation on all basic blocks and fill each block with SSA instructions/values.*

for $block \in blocks$ **do**
  $state \leftarrow createStateVector(block.entryTypeState)$;
  $block.entryState \leftarrow state$;
  for $bytecode \in block.bytecodes$ **do**
    $(state, v) \leftarrow f_{bytecode}(state)$;
    $block.appendInstruction(v)$;
  end
  $block.exitState \leftarrow state$;
end

↪ *Phase 2: Link basic block by forwarding entry states to predecessor's exit states.*

for $block \in blocks$ **do**
  if $isUniqueSuccessor(block)$ **then**
    $forward(block.entryState, predExitState(block))$;
  else
    $forward(block.entryState, \phi(predExitStates(block))$;
  end
end

↪ *Phase 3: Optimize ϕ instructions.*

Algorithm 1: Static SSA Form Construction

6.3.1 Dynamic Control Flow Graphs

We define the term *dynamic control flow graph* to represent an expansion of a static CFG, whereby expansion means a set of transformations of the static CFG. These transformations include speculative execution, tail duplication, loop unrolling and inlining, all of which are implicitly applied during the execution of a program, hence the word *dynamic*. Therefore, in this sense, each recorded trace is a linear expansion of a static CFG, or it can be thought of as a subset of the dynamic CFG of a program's static CFG. Furthermore, a trace region assembled from a set of traces is also a subset of a dynamic CFG. The goal of a trace compiler is to optimally compile a sufficiently representative "hot" subset of the dynamic CFG.

To build such a subset we extend Phase 1 of algorithm 1 so that it performs abstract interpretation on execution traces (algorithm 2 on page 82), and not on the static CFGs as previously described. In our compiler, trace recording always starts at a trace anchor point (p_{anchor}) and usually terminates at the same anchor point, although this is not a requirement and recording may terminate at any other program location. We start out by creating the state vector *state* that will be mutated using abstract interpretation every time a bytecode operation is recorded. This step is exactly the same as in algorithm 1. Next, we create an empty basic block (*block*) to be filled with SSA instructions. This block is created for the current state using a mapping function of the form $getOrCreateBlock(state, equivalence) \Rightarrow block$. This mapping uses an equivalence relation to map state vectors to basic blocks and is the heart of the incremental SSA algorithm. Its job is to maintain a set of basic blocks keyed on their entry state, and index them using an equivalence function.

If a basic block for a given state is not found, one is constructed with the given state vector as its entry state and cached for future evaluations of the mapping function. Following this step, during trace recording, as we observe the execution of individual bytecode operations, we construct and fill the current block with SSA instructions. If a basic block for a given state is found, the taken block has already been recorded. We simply continue tracing, but avoid recording any new instructions in the existing block. If we arrive at a block that has not yet been recorded, we resume recording again. This way we ignore already visited blocks, and explore new execution paths.

At unconditional branches we need to link the branch instruction to the target block, and continue accumulating SSA instructions in this unconditionally taken block. To compute the target block we apply the mapping function to the target state. The target state is the state at the target program counter (PC) location, which we can create by changing the current state to have a new location.

Conditional branches are handled nearly identically, with the exception that two successor blocks need to be computed, and only one of them needs to be selected as the taken block. The selection is made by evaluating the branch instruction, which is possible because we are tracing the execution path of the program and can determine which block was taken as the result of the branch condition.

At call sites, if we decide to inline virtual or interface calls, we insert type guards for the receiving object type and chain the new state vector to the current state vector. Lastly, if we record a bytecode operation for which an exception handler exists, we construct a handler block by using the mapping function along with the *Value* equivalence relation (discussed in section 6.3.2).

During incremental construction, blocks that have not been visited remain empty. These empty blocks are region exit points. Phase 3 of the algorithm 2 on the following page creates transfer instructions that transfer control to another execution mode using the entry state of each exit block.

6.3.2 State Maps & Equivalence Relations

As mentioned previously, state maps are at the heart of our incremental construction algorithm. The reason for this is that they generalize the construction of the trace formation techniques discussed in section 3.2 on page 16. State maps are essentially key-value maps that map state vectors to basic blocks for some equivalence relation. Given an equivalence relation, a state map returns the first block whose state is equivalent to the query state. If no such block exists, a mapping from the query state to the newly constructed block is added to the state map. The following subsuming equivalence relations effectively make tail duplication decisions and are used in our compiler:

Always Non-Equivalent This relation always fails when applied to two state vectors. Although seemingly not useful, when used in algorithm 2 on the next page, this relation allows us to completely unroll the execution of a trace. Using this relation the algorithm

input : A *trace* of bytecode operations

↪ *Phase 1: Initialize the state vector to match the anchor's entry type state and create an entry block for it.*

$state \leftarrow createStateVector(anchor.entryTypeState)$;
$block \leftarrow getOrCreateBlock(state, Location)$;

↪ *Phase 2: Perform abstract interpretation on each bytecode operation observed during tracing. Create and fill blocks with SSA instructions/values.*

for *bytecode* ∈ *trace* **do**
  ↪ *Potentially inline virtual or interface calls by guarding on their receiver's type.*
  if $isCallOperation(bytecode) \wedge shouldInlineCall()$ **then**
    if $isVirtualOrInterfaceCall(bytecode))$ **then**
      $block.appendTypeGuardInstruction()$;
    end
    $state \leftarrow createInlinedStateVector(state, bytecode)$;
  else
    $(state, v) \leftarrow f_{bytecode}(state)$;
    $block.appendInstruction(v)$;
  end
  if $isUnconditionalBranch(v)$ **then**
    $targetState \leftarrow changeStateVector(state, v.target_{PC})$
    $v.target \leftarrow getOrCreateBlock(targetState, Location)$;
    $block \leftarrow v.target$;
  else if $isConditionalBranch(v)$ **then**
    $targetState \leftarrow changeStateVector(state, v.target_{PC})$
    $v.target \leftarrow getOrCreateBlock(targetState, Location)$;
    $followState \leftarrow changeStateVector(state, v.follow_{PC})$
    $v.follow \leftarrow getOrCreateBlock(followState, Location)$;
    ↪ *Evaluate the taken branch target block by inspecting the state of the runtime stack.*
    $block \leftarrow evaluateBranchTarget(v)$;
  end
  ↪ *Evaluate the taken branch target block by inspecting the state of the runtime stack.*
  if $throwsException(v)$ **then**
    $handlerState \leftarrow changeState(state, v.handler_{PC})$
    $v.handler \leftarrow getOrCreateBlock(handlerState, Value)$;
  end
end

↪ *Phase 3: Create transfer instructions for exit blocks.*

for *block* ∈ *blocks* **do**
  if $isEmpty(block)$ **then**
    $block.appendTransferInstruction(block.entryState)$;
  end
end

↪ *Phase 4: Same as phase 2 & 3 in algorithm 1 on page 80.*

Algorithm 2: Incremental SSA Form Construction

will never find an already existing basic block to join control flow back to and instead will create a new basic block each time. A linear trace compiler can be built quite easily using this relation.

Figure 6.3b on page 84 shows the dynamic CFG that is built from the static CFG in

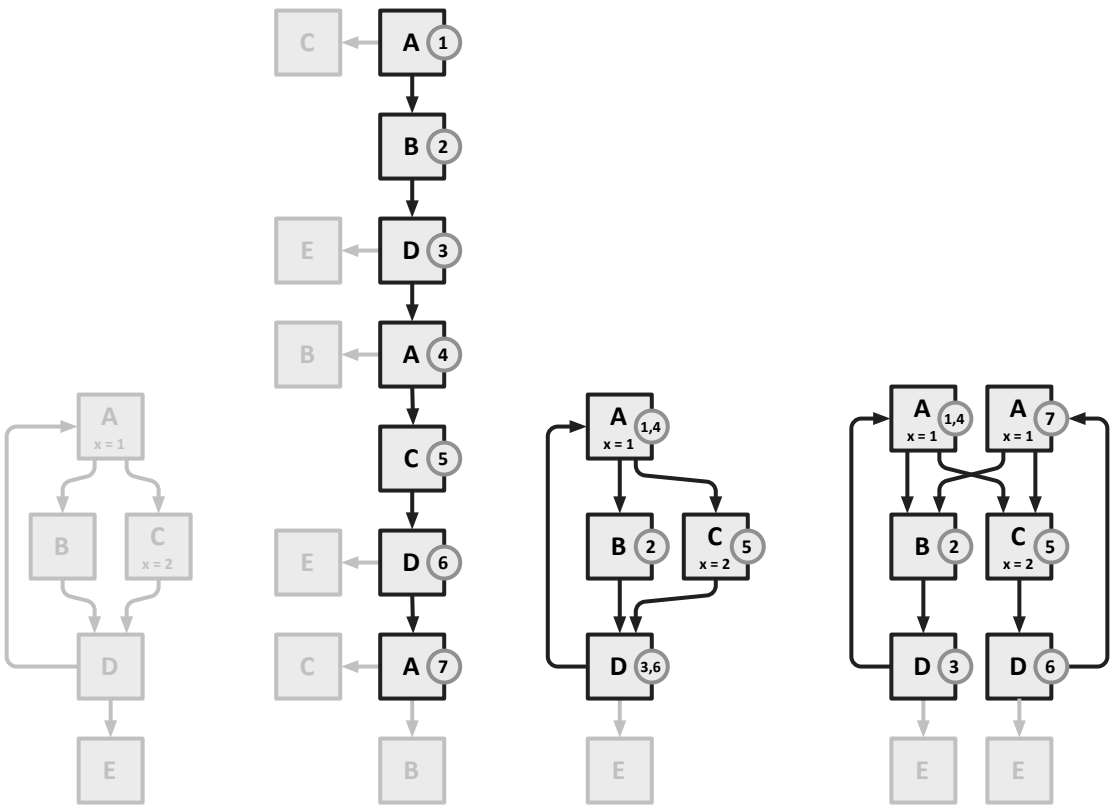
figure 6.3a using this type of equivalence relation given the program trace: {A, B, D, A, C, D, A}. The dynamic CFG is a linear trace with no control flow merge points. The gray nodes represent exit blocks.

With slight modifications, a trace tree compiler can be implemented as well. Trace trees are essentially a collection of linear traces that tail duplicate all basic blocks starting at a given anchor program location, with the constraint that all traces must eventually link back to this location. Therefore, the relation has to be extended to report a single equivalence: the entry state of the loop header. This automatically links the backward branch of the loop back to the (already existing) loop header.

Location Equivalence Two state vectors, a and b, are *location equivalent* if $a^x_{bci} = b^x_{bci} : x \in \{1, \ldots, n\}$ where n is the number of chained inlined states in a and b. In other words, both a and b have the same exact method inlining context and program location. When used in algorithm 2 on the previous page this equivalence relation creates a basic block for every control flow block that is visited during tracing at each inlining context. We use this equivalence to construct trace regions without any tail duplication. Figure 6.3c on the following page shows the dynamic CFG that is built using location equivalence; instead of replicating each basic block, control flow is joined at previously visited basic blocks.

Type Equivalence A state vector a is *type equivalent* with b if $a^x_{bci} = b^x_{bci} \wedge a^x_y \subseteq b^x_y : x \in \{1, \ldots, n\}, y \in \{1, \ldots, m_n\}$ where m_n is the number of values in the chained inlined state n. In other words, the type state of a is compatible with the type state of b. Type equivalence subsumes location equivalence. When applied only to basic types like `int`, `float`, and `Object`, it is in fact redundant in Java because the bytecode verifier ensures that the declared type state at a given program location is always fixed. However, when also taking runtime types of objects into account, this can be used to perform tail duplication based on types observed at runtime. In dynamically typed languages, this technique can be applied to speculate and perform tail duplication based on dynamic runtime types. This is an important optimization used, e.g., in the TraceMonkey JavaScript VM [GES+09].

Value Equivalence Two state vectors, a and b, are *value equivalent* if they are location equivalent and $a^x_{bci} = b^x_{bci} \wedge a^x_y = b^x_y : x \in \{1, \ldots, n\}, y \in \{1, \ldots, m_n\}$. In other words, both a and b have the same exact method inlining context and program location, and the same exact SSA values in the state vector. We use this equivalence to generate exception handler blocks because it allows us to cache and reuse exception handling stubs that share value-equivalent entry states. Figure 6.3d on the next page shows the dynamic CFG that is built using value equivalence. In this example, there are two assignments to variable x, one in block A ($x = 1$) and another in block C ($x = 2$). When block D_3 is visited, the SSA value of x in its state vector is 1, as a result of its previous assignment in block A_1. This however, does not match the state after C_5 is visited, where x is assigned 2, which leads to the construction of two new blocks D_6 and A_7 that expect x to be 2. Although blocks B and C become merge points, they do not merge dataflow and therefore do not require any phi instructions.



(a) Static CFG (b) Non-Equivalent (c) Location Equivalence (d) Value Equivalence

Figure 6.3: Construction of a dynamic control flow graph using equivalence relations.

6.4 Implementation

Until April 2011, Maxine relied an inefficient continuation passing style (CPS) optimizing compiler. Its front end structure was not suitable for trace-compilation and we decided to build our own SSA based optimizing compiler from the ground up. We still however shared the CPS compiler's backend for register allocation and code generation. The previous sections described the incremental SSA construction algorithm used in this compiler.

Starting in April 2011, the Maxine team deprecated the CPS compiler and began using a port of Hotspot's state of the art C1 (client) compiler [KWM+08]. [3] The Java port of C1 (which was previously written in C/C++) is called C1X and is now the primary optimizing compiler in Maxine. In order to coevolve with the new design we had to abandon our previous compiler and adapt to the new C1X compiler. Luckily, unlike the old CPS compiler, the C1X compiler uses an intermediate representation based on SSA form. However, it is still largely designed as a method compiler that operates on statically known control flow graphs. In this section we describe the design of the C1X compiler and how it was adapted to use our incremental SSA construction algorithm.

[3] The Sun Hotspot JVM contains two compilers: the C1 (client) compiler and the C2 (server) compiler.

6.4.1 C1X Compiler Architecture

Kotzmann et al. [KWM+08] provide an excellent description of the design of the Java HotSpot™ Client Compiler for Java 6. The C1X compiler retains C1's original design and includes several improvements and simplifications while porting it to Java. The high level structure of the C1X compiler is shown in figure 6.4 on the following page and is separated in two phases:

Front End : The first phase constructs a high level intermediate representation (HIR) in SSA form via abstract interpretation of bytecode, similar to our own technique. Abstract interpretation is performed by the *HIR Graph Builder* in a single, linear pass over bytecode which is very efficient. During this process a large number of *Level 1 Optimizations* are performed, including: method inlining, constant folding, strength reduction, local value numbering, load elimination, and null check elimination. This is possible because the abstract interpretation is performed in a single, linear pass over bytecode in reverse post-order (i.e., each basic block is processed after all of its predecessors are processed, except for backward edges). Instructions that can be optimized during abstract interpretation are not included in the HIR graph. This allows further, more powerful *Level 2 Optimizations* to operate on a much smaller HIR graph. These optimization include: global value numbering, more powerful null check elimination, control flow optimizations and dead code elimination.

Back End : The second phase transforms HIR into a lower level intermediate representation (LIR) that is conceptually similar to machine code and performs register allocation on it. C1X uses a linear scan register allocator [WM05] to map virtual registers to machine registers, followed by a LIR assembler that emits native code in a straightforward fashion. In this phase, object maps for garbage collection and debug information for deoptimization are also processed and emitted along with the generated code.

6.4.2 C1X Tracing Architecture

Because of the large number of optimizations that occur in C1X's method based HIR Graph Builder, we could not simply shortcut its execution and generate HIR code directly from incremental SSA construction algorithm. Doing so would severely put our trace compiler at a disadvantage when comparing it with a method based compiler. Instead, we had to retrofit the method based HIR Graph Builder to support incremental SSA construction.

Method Graph Builder As mentioned previously, the HIR Graph Builder algorithm processes bytecode basic blocks in a single linear pass, in reverse post-order. The graph builder first builds a skeleton CFG for the method it is processing, and then begins to fill it via abstract interpretation starting with the entry block. At the end of the block, it propagates its exit state to all of its successor blocks and adds each of these to a sorted work list [4] if they haven't already been processed. The algorithm continues with the next block

[4] The work list is sorted on each block's depth first iteration order.

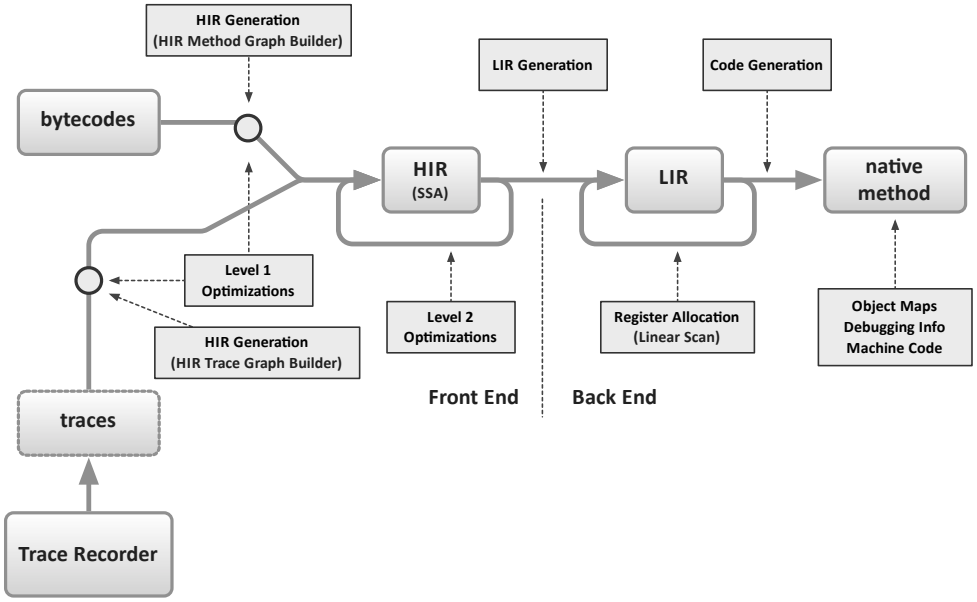


Figure 6.4: Architecture of the C1X compiler.

in the work list until the work list is empty. Because the reverse post-order guarantees that all predecessor of a block are visited before the block itself, each block can assume that its entry state has been computed and is fixed. The only exception is for loop header blocks, in which case the algorithm preemptively inserts ϕ functions in its entry state vector in case a local variable is updated in the loop across a back edge. In the forward direction however, multiple blocks may propagate their exit state to single join block. In which case, exit states are merged with ϕ functions in the join block whenever they do not match.

For example, consider the CFG in figure 6.6 on page 88. A possible reverse post-order for this graph is {A, B, C, E, F, D, G, H, I, J}. Table 6.5 on the following page shows the state propagation through this CFG based on this order. The algorithm starts processing the CFG at block A and then merges its exit state at block B because B is a loop header, the same thing happens again when processing back edge at block G. The algorithm continues at block B where it simply copies the exit state to each of its sucessors, C and D, both of which eventually propagate their exit states to block G. The first to do this is block E which simply copies its state to block G, followed by block F which now has to merge its exit state at block G. State merging involves creating ϕ functions for each of pair of values in the joining state vectors that do not match. Note that the exit state from blocks E, F, and D are all merged at block G before G is ever processed. When processing block G, its exit state is copied to block H and also merged at block B which already has ϕ functions in its entry state. ϕ functions are normally inserted conservatively for all local variables at loop headers. Optionally, the compiler can perform a simple analysis ahead of time to determine which local variables are updated in the loop, and only insert ϕ functions for those local variables at corresponding loop headers. Although superfluous ϕ functions are eliminated by later optimization passes, their introduction early on during the graph building phase inhibits many Level 1 Optimizations and thus it is important to avoid introducing them superfluously.

#	Block	State Propagation	Worklist
1	A	A $\hookrightarrow$ B	{B}
2	B	B $\mapsto$ C, B $\mapsto$ D	{C, D}
3	C	C $\mapsto$ E, C $\mapsto$ F	{E, F, D}
4	E	E $\mapsto$ G	{F, D, G}
5	F	F $\rightsquigarrow$ G	{D, G}
6	D	D $\rightsquigarrow$ G, I	{G, I}
7	G	G $\hookrightarrow$ B, G $\mapsto$ H	{H, I}
8	H	H $\mapsto$ J	{I, J}
9	I	I $\rightsquigarrow$ J	{J}
10	J		

Figure 6.5: State propagation in the Method Graph Builder for the example in figure 6.6 on the following page. Symbol Legend: $\mapsto$ copy state, $\rightsquigarrow$ merge state, $\hookrightarrow$ merge state at a loop header.

During abstract interpretation, Level 1 Optimizations annotate SSA values in various ways. For instance, instructions that allocate objects can be annotated (with attributes) as being `NonNull` since their resulting value is guaranteed to always be not null. Therefore, any operation that dereferences this SSA value, such as a field access or a virtual method invocation does not need to perform a null check. If the object allocation statement, "`Object x = new Object();`" appears in block B, then null checks can be eliminated in all of the blocks that block B dominates: {C, E, F, D, G, H, I and J}. However, if the variable `x` is reassigned in block F to a value that is not marked as `NonNull`, then only blocks: {C, E, D and I} can be optimized. Since there are three possible attributes for variable `x` that can flow into block G, the function ϕ(`NonNull`, `Unknown`, `NonNull`) is necessary to merge dataflow. The attribute of this value is `Unknown` and therefore any block that it propagates to cannot be optimized (G, H and J).

If the object allocation statement appears in block A, block B receives a placeholder ϕ(`NonNull`, `?`) function, due to conservative ϕ function insertion. Even if `x` is never reassigned, the attribute of the ϕ function remains `Unknown` and no optimizations can occur until block G is processed. After block G is processed, the attribute of the now ϕ(`NonNull`, `NonNull`) function is determined to be `NonNull` because no new value of `x` is carried over along the back edge. Subsequent blocks {G, I and J} can benefit from this newly computed attribute, but for the already processed blocks {B, C, E, F, D and G} it is too late. These will have to be optimized at a later time, in a Level 2 Optimization.

Trace Graph Builder The Trace Graph Builder builds control flow graphs incrementally as it processes execution traces. It interacts closely with the Trace Recorder in order to guide the exploration of static control flow graphs. The Trace Graph Builder operates in sync with program execution, meaning it processes basic blocks in execution order. In contrast with the Method Graph Builder, this order is not equivalent to the reverse post-order. Figure 6.7 on the next page shows the operation of the Trace Graph Builder as three traces are processed. The first recorded trace follows the execution path shown in figure 6.8a on page 89, namely {A, B, D, I, J}. The second trace follows the path {A, B, D, G, H, J} which explores two additional blocks {G, H}. The third trace follows the path {A, B, C, E, G, B, D, I, J} which also explores two new additional blocks {C, E} while

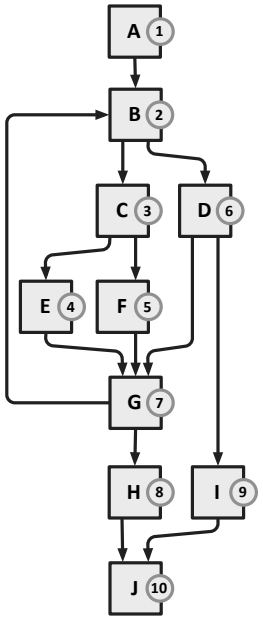


Figure 6.6: Order in which the Method Graph Builder processes basic blocks.

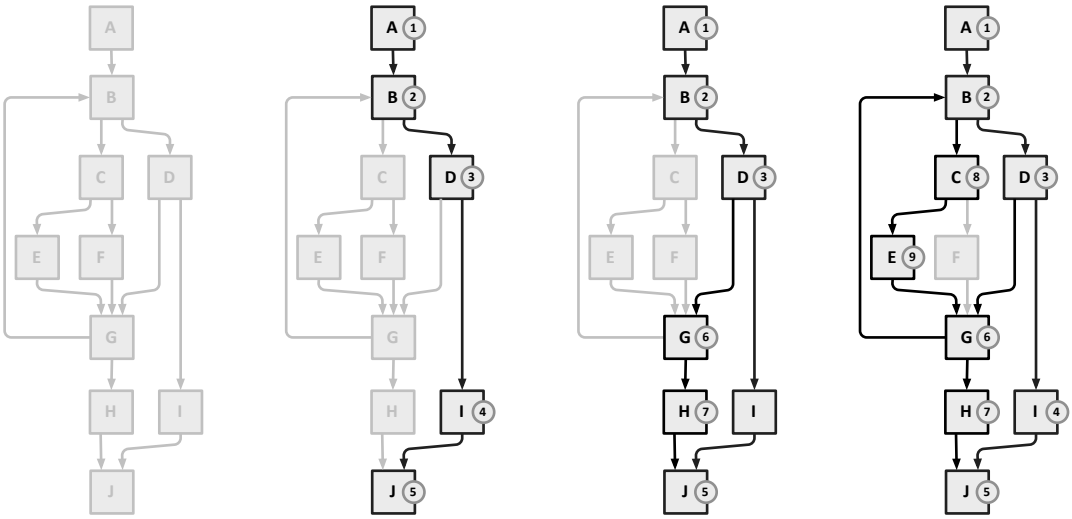


Figure 6.7: Order in which the Trace Graph Builder processes basic blocks. Light gray means unprocessed; dark gray means that the block has been processed in the order indicated by the circled number.

recording one iteration of the loop.

When building a dynamic control flow graph, there is no way of knowing ahead of time, which blocks will be included in the graph and which ones will be left out. This leads us to an interesting situation. On one hand, we would like to capture as much of the

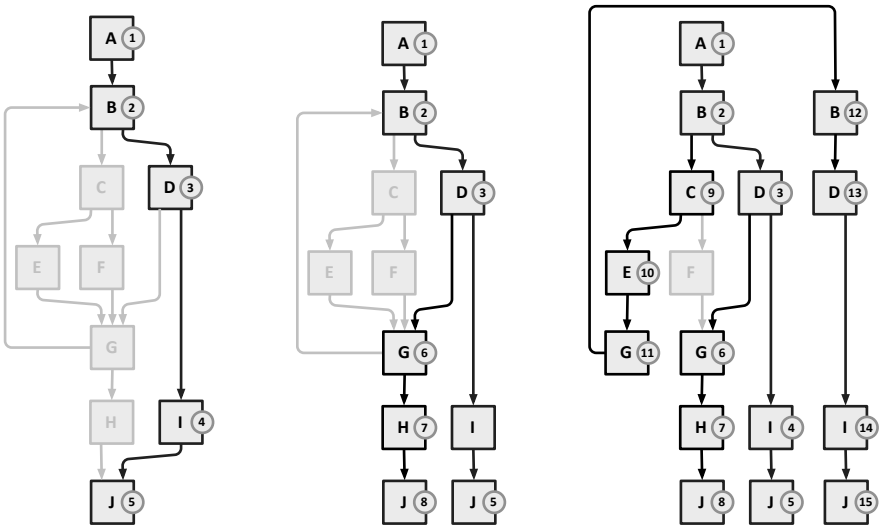


Figure 6.8: Tail duplication in the Trace Graph Builder.

dynamic control flow as possible, thus improving the scope of optimizations and reducing the number of region exit blocks. On the other hand, we would like to minimize the number of join blocks by capturing less of the dynamic control flow, which also improves optimizations. The difficulty is that when we reach a join block for the first time, there is no way of knowing if another trace will be recorded that reaches that same block, but along a different control flow path. For instance, in figure 6.8a when block J is reached, we have the choice of propagating the exit state of block I or introducing placeholder ϕ functions in the expectation of a new trace that reaches J but along a different path, as is the case in figure 6.8b when J is reached via block H. The same thing happens with block G and B in figure 6.8b when new paths are discovered that join control flow. If we introduce placeholder ϕ functions then we prohibit Level 1 Optimizations downstream, if we do not, then we cannot join control flow in the future and must either stop tracing or duplicate the join block. In the Method Graph Builder this is not a problem because all predecessors of a join block are visited before join block itself. Figure 6.7 on the preceding page shows the construction of a dynamic control flow graph where all join blocks receive ϕ functions prematurely thus allows for joining control flow in the future. This type of construction is based on *location equivalence*. We always merge state at program points that are *location equivalent*.

Figure 6.8 shows the construction of a trace control flow graph, based on the original three traces, where all join blocks and downstream blocks are duplicated. This type of construction is based on *value equivalence*. We only duplicate join blocks whenever states are not *value equivalent* (when merging dataflow would require the insertion of ϕ functions). This may cause some blocks to be duplicated but does not necessarily preclude dataflow merge points downstream. The choice of whether to use *location equivalence* or *value equivalence* or a combination of the two, is largely dependent on the application.

Partial Method Inlining Method inlining within C1X compiler only occurs during graph building. During abstract interpretation, at every call site, the graph builder first performs a class hierarchy analysis to determine if it can de-virtualize the call, and then decides whether to inline the method based on several criteria, usually the size of the callee weighted by the current inlining depth. Inlining works by recursing the graph builder into the callee and performing abstract interpretation in the callee context. This inlines the entire control flow graph of the callee into the caller.

In the Trace Graph Builder we have extended this process to support partial method inlining based on execution traces. When reaching a call site, instead of inlining the complete control flow (which we can still do), we ask the Tracer Recorder to trace the execution of the callee. (see section 5.3.2 on page 55 for a discussion on how the Trace Recorder can guide program execution) This way, we only inline a small fraction of the callee, which in turn allows us to inline much deeper. Full inlining is still necessary for semantic reasons in some cases. Some methods in the Maxine VM require that they are completely inlined in order to operate correctly. To support this, we ask the Trace Recorder to skip over the execution of the callee and then use Method Graph Builder to inline its CFG.

Trace Graph Pruning After the trace graph has been sufficiently extended, we prune the graph by eliminating all basic blocks that lead to a trace graph exit. These blocks usually appear whenever trace recording is aborted, due either to an excessive number of blocks being recorded, or to the program reaching a `throw` exception statement. This can leave already constructed parts of the trace graph in a disconnected state. Pruning eliminates these blocks, their compilation would not benefit their execution since they would lead to a trace exit. After pruning, we introduce transfer instructions in all off-trace blocks and compile the control flow graph.

Chapter 7

Applications of Trace-Based Compilers

7.1 Tracing in Dynamic Languages

Dynamically typed programming languages have emerged as the paradigm of choice for client-server computing. On the server side, Python, Ruby, and PHP are used by today's largest websites, while JavaScript has emerged as the dominant language on the client side.

This development has caught the developers of just-in-time compilers somewhat off-guard. In the original roadmap for the web, generally accepted until quite recently, executable content was supposed to be provided by *statically typed* languages such as Java Virtual Machine Language and the .NET Common Intermediate Language. Over the past decade, powerful just-in-time compiler frameworks have been developed that handle these statically typed languages surprisingly well.

Unfortunately, in the very recent past, there has been almost no activity in executable web content based on statically typed intermediate languages. Instead, there has been an explosion of executable content in dynamically typed languages. A typical web browser today executes orders of magnitude more JavaScript than Java, both in code volume as well as in dynamic instructions. Ironically browsers until recently contained highly optimized and ambitious JVML execution engines and only mediocre JavaScript execution engines.

Implementations of JavaScript such as Tamarin [Mozb] and Rhino [Moza] have begun to use just-in-time compilation, these compilers perform significantly worse than their counterparts for statically typed languages such as JVML. The main reason for this discrepancy is the untyped nature of JavaScript. While a JVML program contains primitive operations such as `iadd` that tell the virtual machine directly that two integers need to be added (and which can be translated directly into a corresponding machine instruction), every operation in a JavaScript program is late-bound, i.e., depends on the dynamic type of the argument which can change even between successive executions of the exact same instruction. Hence, a costly dynamic dispatch is needed that cannot be easily optimized away.

One optimization technique is to use type specialization [CU89] in order to specialize code for the most common dynamic types of receiver arguments. For example, we can create a special version of a program fragment in which we assume that a certain variable is an integer. In that special version, we can then map an `add` operator directly to integer addition, and can even perform arithmetic simplifications on the code. In theory, every possible combination of types for all variables involved might occur, but compilers use *type inference* to reduce the number of such type combinations quite effectively. However, there are always cases in which a concrete type cannot be inferred statically.

We have explored a new technique that leverages an existing trace-based compiler in order to automatically generate type specialized code fragments for dynamically typed languages.

7.1.1 Static Type Inference

In traditional compilers for dynamically typed languages, static type inference is used to predict the runtime type of untyped variables. Figure 7.1 on the next page shows a simple JavaScript program. The value of the loop variable `i` is accumulated in variable `x` until `x == b`, at which point `x` is prefixed by the string "`number:`". This causes `x` to become a string and further executions of the statement `x = x + i` to perform string concatenation rather than number addition. Furthermore, variable `i` starts out as a number (constant integer `0`) and is incremented by the integer constant `1` every loop iteration (`i++`) until `i < a`. The JavaScript specification defines that numbers must be represented using the double-precision 64-bit IEEE 754 format ($2^{64} - 2^{53} + 3$ possible values). However, from a performance perspective, numbers can be represented much more efficiently using integers as long as they are small enough to fit within the integer range (-2^{31} to $2^{31} - 1$), and if they exceed this range, they can be converted into the less efficient double-precision format. In our example, if the value of `a` is grater than 2^{31} then the type of `i` will change to `double`.

To infer this type information, many compilers for dynamically typed languages use iterative data-flow analysis. First, all known types are flagged. The types of constants, for example, are usually known, and thus in our example we would flag the results of the initial assignments to `x` and `i` as integers. Through subsequent analysis it can then be discovered whether these initial type assignments are preserved by every operation that executes on the variables.

To guarantee correct semantics, this type inference process must be *precise*. If the analysis infers an integer type for a variable, for example, but that variable can become a floating point value at some point, the generated machine code would not be able to handle this case correctly since it would use an integer register to hold the value.

The precision requirement often prevents generation of the most efficient machine code. In JavaScript, for example, the addition of two integers actually returns a floating point DOUBLE value if the additions overflow the range representable by integers. This significantly complicates static type inference, since we can no longer safely infer that the result of an addition of two integers will always produce an integer. It will produce an integer only as long no overflow occurs. Otherwise, a floating point value will be returned.

In these cases, when static type inference cannot precisely predict a single type for a variable, it assigns the `any` type, which at runtime is implemented using a tagged value that can store any type. Operating on variables of this type is of course significantly more expensive at runtime since the appropriate operation has to be selected dynamically depending on the dynamic type of the variable at that particular point in the program execution. Thus, using standard static type inference techniques, it would be impossible to compile this loop into efficient code.

```
1 var x = 0;
2 for (var i = 0; i < a; i++) {
3     x = x + i;
4     if (x == b) {
5         x = "number:" + x;
6     }
7 }
```

Figure 7.1: A simple accumulation loop in JavaScript. Through the initial assignment `x = 0`, the variable `x` has the type `int` at the loop entry. Variable `x` will remain of type `int` until it eventually exceeds the value range of the integer type. Moreover, when `x == b`, `x` will be changed to a string. Further `x = x + i` statements will result in concatenating a textual representation of `i` to the string `x`.

7.1.2 Deferred Runtime Type Inference

We use an alternative approach for the efficient compilation of dynamically typed programming languages. Instead of attempting to infer type information statically ahead-of-time, we translate the dynamically typed code into a statically typed JVML code in which the semantics of dynamic typing are modeled precisely by a runtime library (figure 7.9 on page 104). For every operation (i.e., `+`), the runtime library considers *both* operands to select the correct semantics to be applied. This is a common technique used by many virtual machines that run on top of the JVM, such as Rhino [Moza] or Jython [Jyt]. At first sight, this shifts the burden of dynamic typing entirely onto the runtime system. However, in conjunction with trace-based compilation, this method actually results in better code because the overhead of dynamic type checks can be optimized away in many cases due to speculative execution nature of the underlying trace-based compiler.

When translating JavaScript code into the statically typed JVML representation, all variables are declared as instances of `JsValue`, and all operations are translated to method calls in the `JsValue` class. The `JsValue` class in figure 7.3 on page 100 encodes all possible JavaScript values by using common technique called tagged union where a tag field explicitly indicates the type of the overlapping value storage area. Since Java does not support unions, we use a field (`type`) to indicate the type the value stored in one of three fields in the `JsValue` class: `intValue`, `doubleValue` and `objectValue`.

Every JavaScript operation is implemented as a method in this class. For instance, the addition operation is implemented by the `void add(JsValue right) {...}` where the first (implicit `this`) operand is also the destination (the statement `x += y` is implemented

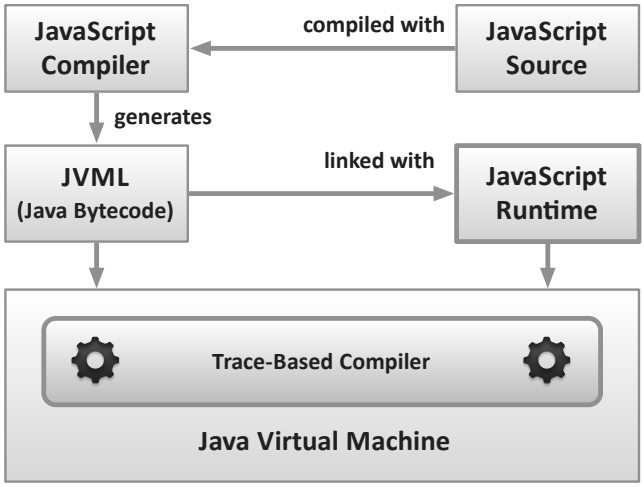


Figure 7.2: Architecture of our JavaScript VM layered on top of a JVM using a Trace-Based Compiler.

using `x.add(y)`). The dynamic `add` method must perform the operation for all possible permutations of types, in our example: {`int`, `double`, `string`} × {`int`, `double`, `string`}, 9 permutations. The `add` method selects the appropriate operation using a several nested type check statements. The order in which types are checked is also important. Selecting the `int` × `int` combination can be done with only two type checks, while selecting the last combination, `string` × `string` requires a total of 6 type checks (3 for each outer and inner `if ... else` statements). Figure 7.3 on page 100 shows the implementation of some of these cases:

- `int` × `int`: even if both of the operands are of type `int` their sum may overflow the integer range. We check for overflow by performing the addition operation on longs and then checking whether the sum is within the integer range, if it is not, then we convert the destination operand to a `double`.
- `int` × `string`: JavaScript semantics dictate that the result of the operation is the concatenation of the string representation of the left integer operand with the right operand.

Figure 7.4 on page 101 shows the JVML code that is generated for the simple accumulator loop in figure 7.1 on the preceding page. It contains 6 calls to dynamically typed JavaScript operations: `less`, 3 × `add`, `equals` and `copy`. Each of these operations requires between 2 ~ 6 type checks to dispatch to the appropriate type operation. In total, between 12 ~ 32 type checks are required to select the appropriate operations to execute each loop iteration. Reducing the number of type checks is critical to improving the performance of dynamic languages.

A standard JVM optimizing compiler has trouble optimizing many of these type checks for several reasons:

- In order to perform optimizations, the compiler needs to inline the methods in the `JsValue` class. Doing so, gives the compiler contextual information that it can sometimes use to eliminate redundant type checks. This, however, is not possible because these methods are quite large. Inlining them all would significantly reduce compilation speed and overall performance. Figure 7.5 on page 101 shows the program in figure 7.4 on page 101 with only one of its `add` operations inlined at line 5, inlining the remaining 5 operations would cause the source code listing to span several pages, and only as a result of a small JavaScript program.

- In figure 7.5 on page 101, although the compiler has inlined the method, it cannot perform any optimizations since it cannot prove anything about the type of `x`. It starts out as `int` but may become `double` or `string` at a later time.

A trace-based compiler is perfectly suited for this type of program workload but for the exactly opposite reasons:

- A trace-based compiler can use partial method inlining to chose only the method fragments that are involved in a given type configuration. Doing so, the trace-compiler can afford to inline a much larger number of methods.

- A trace-based compiler is speculative in nature. It produces specialized code that is suited for the current phase of the program it executes. Since it only compiles the most frequently executed control flow paths, the trace-compiler essentially specializes the code for the most common type configuration. In figure 7.1 on page 93, the most common type configuration is when `i` and `x` are `int` (provided `a` is relatively small and `x` is not likely to equal `b`). Figure 7.6 on page 102 shows an unoptimized trace recorded by the trace-based compiler for the example program where every method is inlined. Program paths that were not explored during trace recording receive `TRANSFER` instructions which transfer execution to type generic code. As far as the optimizing compiler is concerned, `TRANSFER` terminate control flow. They act like `return` or `throw` statements. This means that the control statement `if (c) { TRANSFER } else {...}` has no join node, and teaches the compiler that all code downstream can assume that `c` is `false`. This has very significant ramifications for compiler optimizations, it allows the compiler to remove redundant downstream type checks. figure 7.7 on page 103 shows the resulting code after redundant checks were eliminated.

- Furthermore, in figure 7.6 on page 102 variables `x` and `i` are always integers, the integer overflow cases are not captured by the recorded trace. This means, that as far as the optimizing compiler is concerned, it can simply assume that `x` and `i` are always integers. Additionally, the compiler can also detect that variables `a` and `b` are loop invariant, and can be hoisted outside of the loop. After these transformations, the compiler produces the highly optimized code in figure 7.8 on page 104.

- Aside from speculative control flow optimizations, a trace-based compiler can also speculate on concrete runtime values. For instance, during trace recording, if we observed that the value of `a` is less than `1000` then we can take advantage of this information. We can introduce a speculative value guard outside of the loop and

eliminate the last overflow check, since it's not possible for a value less than 1000 to overflow when incremented by 1.

The truly remarkable thing is that these optimizations can occur without any help from the dynamic language implementor. All that is required is that the host virtual machine includes a trace-based compiler. As an end result, in some cases a trace-based compiler can generate code that is sometimes as efficient as that for statically-typed languages.

7.2 Trace-Based Compilers for the CLR

We first explored the idea of layering dynamic language virtual machines on top of trace-based compilers in the J^{2}VM system [CBY+07] at the University of California, Irvine. Later, we revisited it in more detail in the SPUR [1] project [BBF+10] at Microsoft Research. In this section we explore the design challenges of a trace-based compiler for the CLR.

The SPUR system is similar in structure to our original design, with the exception that it does not use the JVM as its underlying host VM. Instead, it runs on top of Microsoft's Common Language Runtime (CLR). The CLR uses a common intermediate language called CIL and which flexible enough to support a large variety of programming languages: C#, Visual-Basic, F#, C++, etc. In contrast to Java bytecode, CIL is a much richer and lower level instruction set. In particular, it includes support for *structs* and *reference* types. The SPUR system uses structs to model JavaScript values (tagged unions) and references to pass these structs by-reference to JavaScript runtime library functions. Both of these features significantly increase the complexity of the underlying trace-based compiler.

Structs Because Java does not support structs, tagged unions have to be boxed in `JsValue` objects. This technique has several negative side effects:

- Accessing a value requires one additional level of indirection.
- Temporary values are allocated on the heap and have to be garbage collected.
- Value representation is not compact; each `JsValue` object incurs the additional memory overhead its object header.

Structs, on the other hand, have none of these drawbacks; Figure 7.10 on page 105 contrasts the size of JsValues in Java (40 bytes) with JsValues in CIL (20 bytes) which are half as large (assuming a 64-bit word size); partly due to the fact that in CIL, non-object fields within structs can be overlaid, similar to C/C++ unions. [2] This is used to overlay the `intValue` and `doubleValue` fields in order to achieve a more compact representation.

[1] I contributed the trace-based optimizing compiler to the SPUR system during an internship at Microsoft Research in 2009.

[2] It is not possible to overlay reference fields with data fields because the CLR uses an exact garbage collector; it needs to distinguish between addresses, and data values that happen to look like addresses.

Structs present a difficulty to the trace compiler's SSA graph builder because they behave like collections of local variables. The graph builder must capture the dataflow passing through individual struct fields. Conceptually, any time a struct field is updated, the value of the entire struct is changed; thus a new SSA value is needed to represent the struct's new state. To encode struct updates, we use the pseudo update instruction $set(struct, field, value)$ which evaluates to $struct.field = value$ (pseudo instructions do not appear in the instruction stream). For example, the value of a struct s_0 after the following sequence of assignments $\{s.f = a,\ s.f' = b,\ s.f = c\}$ can be encoded as $s_3 = set(set(set(s, f, a), f', b), f, c)$. Similarly, the value of a field f is defined by the the pseudo access instruction $get(struct, field)$ (or $struct.field$). In the previous example, the value of f' is given by $get(s_3, f') = b$, which can be evaluated by walking the chain of update functions until an update to f' is observed. Encoding the value of structs this way allows the optimizing compiler to perform aggressive optimizations on JavaScript values which are represented using structs.

Modeling structs in SSA form also means that they are never in a materialized form, and must be created (materialized) on demand whenever their concrete state is needed. Materialization is the process of creating a concrete struct by applying the sequence of update functions to a source struct. Such cases include transfers from trace code to the baseline compiler, or whenever they are passed in as arguments to functions.

The trace-based compiler tries to defer struct materialization until trace transfer points, i.e., structs only come into existence when control leaves the trace. This however requires that all arguments of update functions are kept live until the materialization site which extends live ranges and can negatively impact the register allocator.

The situation is further complicated by the fact that structs can be nested and that fields may overlap in memory, and furthermore by cyclic dependencies between struct assignments and ϕ functions at join nodes that merge struct states.

References In order to support passing arguments by-reference, the common intermediate language includes support for reference types. [3] References can refer to local variables, or to fields of structs or heap allocated objects and have several properties:

- In C# for example, references are created using the ref keyword. The reference itself cannot be manipulated directly; it can only be passed in as an argument (e.g. `foo(ref a)` passes a reference to `a` in function `foo()` which can modify the original variable `a`).
- References can never survive the scope of the activation record in which they are created, nor can they escape into the heap. This implies, that during SSA graph building, as long as the reference was not passed in as an argument, we can always trace it back to the variable it refers to. This allows us to track dataflow via reference assignments. References that are passed in as arguments cannot be tracked; fortunately, they cannot possibly alias any local variables, and thus do not interfere

[3] References are typed and known as ByRefs in CIL terminology.

with local references. They can, however, alias variables in the caller frame, or local references that point to fields of heap objects.

- Since SSA form only deals with values and not their storage locations; the concept of references to variables does not fit well into this model, especially when variables may be aliased by one or more references. During SSA graph building, we use pseudo reference instruction ($ref(variable)$) to represent references to variables. Modeling references this way (much like structs) means that they are never in a materialized form, and must be created on demand whenever concrete storage locations are needed.

  Consider the following example simple program: "`int a = 42; foo(ref a); print(a);`". If we were to inline the `foo()` function, we could determine whether variable `a` is reassigned and thus given a new SSA value. If we do not inline the `foo()` function then we cannot determine this fact and have to be conservative. Moreover, we have to pass the address of variable `a` to the `foo()` function. To do this, we create a temporary storage location using two instructions: $v = box(r)$ which creates a storage location with the value v and returns its address, and $v = unbox(r)$ which performs the opposite operation. Using these, we can temporarily box the SSA value a_0, pass its references to `foo()` and read its possibly updated value after the call; the resulting SSA program being: $a_0 = 42$; $r_0 = box(a_0)$; $foo(r_0)$; $a_1 = unbox(r_0)$; $print(a_1)$;.

  The situation is further complicated by the fact that references may have to be materialized at trace transfer points, which may be in inlined contexts that are several frames deep. In such cases, the trace transfer code must reconstruct the several baseline frames as well as compute and write back concrete reference values pointing to stack allocated variables that are part of the newly constructed baseline frames.

- The trace-based compiler focuses on frequently executed method fragment. Unfortunately, by doing this, it ignores code that is infrequently executed but which may be beneficial to optimizations nonetheless. For instance, code that is outside of a frequently executed region may take several references to local variables, information which would help the trace-compiler reason about dataflow. Unfortunately, since this code is outside of the trace compiler's sphere of influence, it has to be conservative in regards to aliasing information, just as it must with references that are passed in as arguments.

7.2.1 Optimizations

Aside from standard textbook optimizations, the SPUR trace-based compiler performs several advanced optimizations:

- *Thread Model & Store-Load Propagation*: Although the CLR supports multi-threading, the compiled JavaScript code does not make any use of it. We use this property, to propagate stores to loads within heap allocated propagation objects since no intervening thread of execution can mutate the object between a store and a subsequent load.

- *Guard Elimination*: We eliminate redundant guards and strengthen upstream guards as described in section 4.2.3 on page 33.
- *Alias Guards*: Alias analysis is important for a variety of optimizations; store-load propagation being one of them. Aliasing information is more accurate if we can ensure that references are not aliasing. We can do this by inserting alias guards.
- *Delayed Computation*: Struct materialization allows us move struct updates from the fast path, and sink them in transfer blocks.
- *Invariant Code Motion*: We hoist guards and loop invariant operations above the loop.
- *Annotations*: Because the JavaScript runtime is implemented in C#, we can use CIL annotations to hint to the trace-based compiler that certain objects are constant, or that functions have no side effects.

For copyright reasons, refer to [BBF+10] for an evaluation of the SPUR system.

```
class JsValue {
    Type type; int intValue; double doubleValue; Object objectValue;

    JsValue(Type type, int value)    { this.type = type; intValue = value; }
    JsValue(Type type, double value) { this.type = type; doubleValue = value; }
    JsValue(Type type, Object value) { this.type = type; objectValue = value; }

    JsValue add(JsValue right) {
        if (type == Type.Int) {
            if (right.type == Type.Int) {
                long result = (long)intValue + (long)right.intValue;
                if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
                    // If result overflows the Int range then convert to Double.
                    doubleValue = result;
                    type = Type.Double;
                } else {
                    intValue = (int)result;
                }
            } else if (right.type == Type.Double) { ...
            } else if (right.type == Type.String) {
                objectValue = Integer.toString(intValue) + (String) right.objectValue;
                type = Type.String;
            }
        } else if (type == Type.Double) {
            if (right.type == Type.Int) { ...
            } else if (right.type == Type.Double) {
                doubleValue += right.doubleValue;
            } else if (right.type == Type.String) { ... }
        } else if (type == Type.String) {
            if (right.type == Type.Int) { ...
            } else if (right.type == Type.Double) {
                objectValue = (String) objectValue + (String) right.objectValue;
            } else if (right.type == Type.String) { ... }
        }
        return this;
    }

    boolean less(JsValue right) {
        boolean less = false;
        if (type == Type.Int) {
            if (right.type == Type.Int) {
                less = intValue < right.intValue;
            } ...
        } ...
    }

    boolean equals(JsValue right) {
        boolean equals = false;
        if (type == Type.Int) {
            if (right.type == Type.Int) {
                equals = intValue == right.intValue;
            } ...
        } ...
        return equals;
    }

    public void copy(JsValue value) {
        type = value.type;
        intValue = value.intValue;
        doubleValue = value.doubleValue;
        objectValue = value.objectValue;
    }
}
```

Figure 7.3: Partial implementation of `JsValue` in Java.

```
JsValue x = new JsValue(Type.Int, 0);
JsValue i = new JsValue(Type.Int, 0);

while (i.less(a)) {
    x.add(i);
    if (x.equals(b)) {
        x.copy(new JsValue(Type.String, "number:").add(x));
    }
    i.add(new JsValue(Type.Int, 1));
}
```

Figure 7.4: Statically typed JVML code generated for the JavaScript example code in figure 7.1 on page 93. (For readability we show the Java source code equivalent for the generated JVML code.)

```
JsValue x = new JsValue(Type.Int, 0);
JsValue i = new JsValue(Type.Int, 0);

while (i.less(a)) {
    // Inlined: x.add(i);
    if (x.type == Type.Int) {
        if (i.type == Type.Int) {
            long result = (long)x.intValue + (long)i.intValue;
            if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
                // If result overflows the Int range then convert to Double.
                x.doubleValue = result;
                x.type = Type.Double;
            } else {
                x.intValue = (int)result;
            }
        } else if (i.type == Type.Double) { ...
        } else if (i.type == Type.String) {
            x.objectValue = Integer.toString(x.intValue) + (String) i.objectValue;
            x.type = Type.String;
        }
    } else if (x.type == Type.Double) {
        if (i.type == Type.Int) {
        } else if (i.type == Type.Double) {
            x.doubleValue += i.doubleValue;
        } else if (i.type == Type.String) { ... }
    } else if (x.type == Type.String) {
        if (i.type == Type.Int) {

        } else if (i.type == Type.Double) {
            x.objectValue = (String) x.objectValue + (String) i.objectValue;
        } else if (i.type == Type.String) { ... }
    }
    if (x.equals(b)) {
        x.copy(new JsValue(Type.String, "number:").add(x));
    }
    i.add(new JsValue(Type.Int, 1));
}
```

Figure 7.5: Code in figure 7.4 with only one inlined `add` method call.

```
1  JsValue x = new JsValue(Type.Int, 0);
2  JsValue i = new JsValue(Type.Int, 0);
3
4  while (true) {
5      if (i.type == Type.Int) {
6          if (a.type == Type.Int) {
7              if (i.intValue >= a.intValue) {
8                  break;
9              }
10         } else { /* TRANSFER */ }
11     } else { /* TRANSFER */ }
12
13     if (x.type == Type.Int) {
14         if (i.type == Type.Int) {
15             long result = (long)x.intValue + (long)i.intValue;
16             if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
17                 /* TRANSFER */
18             } else { x.intValue = (int) result; }
19         } else { /* TRANSFER */ }
20     } else { /* TRANSFER */ }
21
22     boolean equals = false;
23     if (x.type == Type.Int) {
24         if (b.type == Type.Int) {
25             equals = x.intValue == b.intValue;
26         } else { /* TRANSFER */ }
27     } else { /* TRANSFER */ }
28
29     if (equals) { /* TRANSFER */ }
30
31     JsValue right = new JsValue(Type.Int, 1);
32     if (i.type == Type.Int) {
33         if (right.type == Type.Int) {
34             long result = (long)i.intValue + (long)right.intValue;
35             if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
36                 /* TRANSFER */
37             } else {
38                 i.intValue = (int)result;
39             }
40         } else { /* TRANSFER */ }
41     } else { /* TRANSFER */ }
42 }
```

Figure 7.6: Unoptimized trace recorded for the program in figure 7.4 on the preceding page.

```
JsValue x = new JsValue(Type.Int, 0);
JsValue i = new JsValue(Type.Int, 0);

while (true) {
    if (i.type == Type.Int) {
        if (a.type == Type.Int) {
            if (i.intValue >= a.intValue) {
                break;
            }
        } else { /* TRANSFER */ }
    } else { /* TRANSFER */ }

    if (x.type == Type.Int) {
        long result = (long)x.intValue + (long)i.intValue;
        if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
            /* TRANSFER */
        } else { x.intValue = (int) result; }
    } else { /* TRANSFER */ }

    boolean equals = false;
    if (b.type == Type.Int) {
        equals = x.intValue == b.intValue;
    } else { /* TRANSFER */ }

    if (equals) { /* TRANSFER */ }

    long result = (long)i.intValue + 1;
    if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
        /* TRANSFER */
    } else {
        i.intValue = (int)result;
    }
}
```

Figure 7.7: Optimized trace for the program in figure 7.4 on page 101, redundant type checks are eliminated.

```
JsValue x = new JsValue(Type.Int, 0);
JsValue i = new JsValue(Type.Int, 0);

if (a.type != Type.Int) { /* TRANSFER */ }
if (b.type != Type.Int) { /* TRANSFER */ }

while (true) {
    if (i.intValue >= a.intValue) {
        break;
    }

    long result = (long)x.intValue + (long)i.intValue;
    if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
        // TRANSFER
    } else { x.intValue = (int) result; }

    if (x.intValue == b.intValue) {
         /* TRANSFER */
    }

    long result = (long)i.intValue + 1;
    if ((result < Integer.MIN_VALUE) || (result > Integer.MAX_VALUE)) {
        /* TRANSFER */
    } else {
        i.intValue = (int)result;
    }
}
```

Figure 7.8: Further optimized trace for the program in figure 7.4 on page 101 after loop invariant type checks are hoisted.

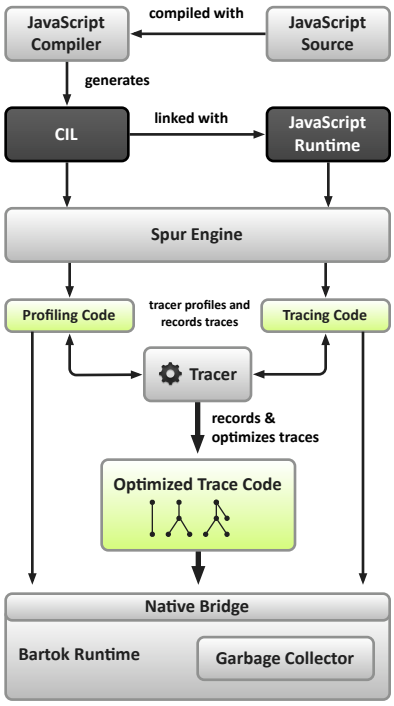
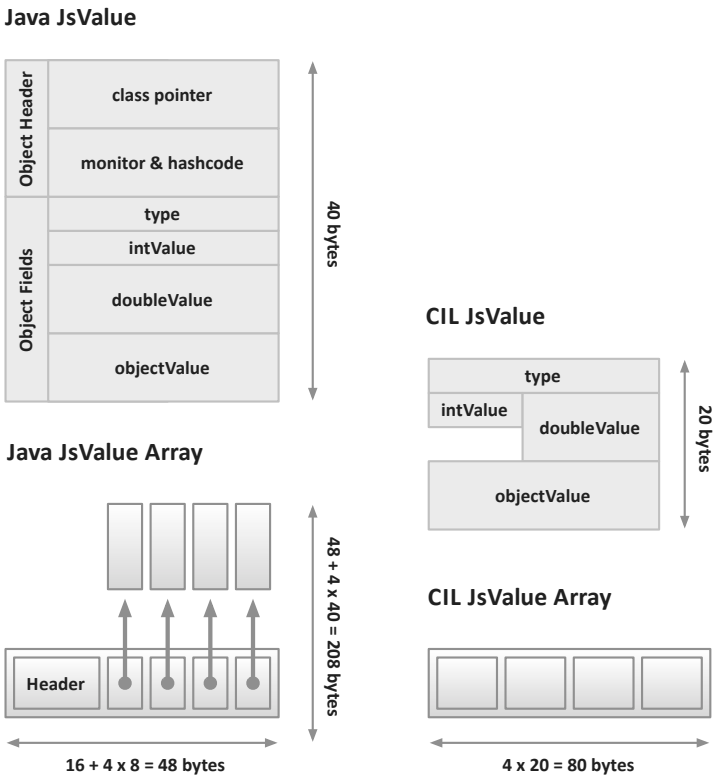


Figure 7.9: Architecture of SPUR system.



(a) Java JsValue (b) CIL JsValue

Figure 7.10: Java vs. CIL representation of individual JsValues and arrays of JsValues. Java version is ≈ 3 times larger due to reference indirection and object overhead.

Chapter 8

Related Work

Tracing is a well established technique for dynamic profile-guided optimization of native binaries. Bala et al. [BDB00] introduced tracing as a method for runtime optimization of native program binaries in their Dynamo system. They used backward branch targets as candidates for start of a trace, but did not attempt to model loops containing multiple alternative paths in a single data structure. Zaleski et al. [ZBS07] used Dynamo-like tracing in order to achieve inlining, indirect jump elimination, and other optimizations for Java. Their primary goal was to build an interpreter that could be extended to a tracing VM.

Gal et al. [GPF06] proposed to build dynamic compilers in which no CFG is ever constructed, and no source code level compilation units such as methods are used. Instead, runtime profiling is used to detect frequently executed cyclic code paths in the program. The compiler then records and generates code from dynamically recorded *code traces* along these paths. It assembles these traces dynamically into a tree-like data structure that covers frequently executed (and thus compilation worthy) code paths through hot code regions. The absence of control flow merge points in this tree-based representation greatly simplifies optimization algorithms and results in quicker optimization passes when compared to compilers that use traditional CFG-based analysis.

Gal et al. [GES+09] extended the previous work on trace-based compilation for Java and built a production-level traced-tree-based VM (TraceMonkey) for JavaScript, currently shipping in the Mozilla Firefox Browser.

Guo et al. [GP11] provide a formal definition of trace recording and execution. They show that a different set of optimizations are sound in a traditional compiler and a trace-based compiler. For example, dead code elimination is unsound in a trace-based compiler because results of seemingly dead code might be used after a transfer out of the trace occurs. This fosters our believe that trace-based compilers require a re-thinking of classical optimizations.

Whaley [Wha01] uses *partial-method compilation* to reduce the granularity of compilation to the sub-method level. His system uses profile information to detect never or rarely executed parts of a method and to ignore them during compilation. If such a part gets executed later, execution continues in the interpreter. Compilation still starts at the beginning of a method.

Similarly, Suganuma et al. [SYN06] propose *region-based compilation* to overcome the limitations of method-based compilation. They use heuristics and profiles to identify and eliminate rarely executed sections of code. In combination with method inlining, they try to group all frequently executed code in one compilation unit, but to exclude infrequently executed code. If an excluded code part must be executed, they rely on re-compilation and on-stack-replacement (OSR). Our trace-based compilation reaches this goal without requiring complex heuristics. They observed not only a reduction in compilation time, but also achieved better code quality due to rarely executed code being excluded from analysis and optimization.

Merrill et al. [MH08] present a solution, implemented on the Jikes RVM, for selecting trace fragments within a VM without an interpreter. Their system compiles each method into two equivalent binary representations: a low-fidelity region with counters to profile hot loops and a high-fidelity region that has instrumentation to sample every code block reached by a trace. When a hot loop has been identified, the low-fidelity code transfers control to the high-fidelity region for trace formation. Once a trace has been formed, execution jumps back to the appropriate low-fidelity region. In their system, profiling and trace formation happens at the machine code level, not at the bytecode level as is the case in our system. Unlike our system, their system assembles traces by stitching together machine level basic blocks that were previously compiled with the whole method just-in-time compiler.

The construction of SSA form for static compilers is already well studied: Starting with the original algorithm of Cytron et al. [CFR+91], several extensions and improvements have been presented [BP03, DR05]. However, all these algorithms focus on static compilation where the complete CFG is available ahead of time. Because trace recording and the extension of trace regions happens are interleaved at runtime, these algorithms are not applicable for our use case. Closer related to the algorithm presented in this dissertation are SSA form construction techniques used in current just-in-time compilers: The Java HotSpot™ VM uses an algorithm that conservatively adds phi functions at loop headers and simplifies unnecessary phi functions later on [KWM+08]. The simplification is formally proven correct in [AH00]. We extend this principle and allow phi functions to be inserted conservatively at all places that could be extended later to be control flow join points. Unnecessary phi functions are later simplified by the compiler.

Related to our work on Maxpath, Inoue et al. [IHWN11] present a trace-based compiler retrofitted from a method-based compiler in the production quality IBM J9 JVM. Unlike our approach, their system relies on an interpreter-based baseline execution environment.

Chapter 9

Evaluation

To evaluate the performance and runtime characteristics of Maxpath we have chosen three benchmark suites: Java Grande[MCH99], Spec 2008[Sta08] and DaCapo 2006[BGH+06]. These include a wide variety of benchmark applications, ranging from small computational kernels, to large programs representative of real world applications. All experiments were performed on a Mac Pro with 2 Quad-Core Intel Xeon Processors clocked at 2.8 GHz and 10 GB RAM running MacOS X 10.6. Each benchmark has a self timing mechanism, and these are the results that are presented here.

9.1 Benchmarks

Java Grande Benchmark Suite We have chosen a set of 11 benchmark applications from Java Grande [1] Section 2 & 3. Table 9.1 on the next page describes each Java Grande benchmark in detail.

[1] Java Grande additionally includes a set of benchmarks (Section 1) which consist of very small micro-benchmarks for several Java features; these numbers are not included in our evaluation as they are not particularly interesting.

Mode	Description
Compiler Compiler	Compilation of the javac compiler.
Compiler Sunflow	Compilation of the sunflow benchmark.
Compress	File compression using the LZW compression algorithm.
Crypto RSA	Encryption / decryption using RSA.
Crypto AES	Encryption / decryption using AES.
Crypto Signverify	Signature verification.
Derby	Databse in Java emphasizing BigDecimal computations.
Mpegaudio	Mp3 audio decoder in Java emphasizing floating-point computations.
Scimark FFT	Fast Fourier Transform (small 512KB, large 32MB)
Scimark LU	LU Factorization (small 512KB, large 32MB)
Scimark SOR	Successive Over-Relaxation (small 512KB, large 32MB)
Scimark Sparse	Sparse Matrix Multiplication (small 512KB, large 32MB)
Scimark Monte Carlo	Monte Carlo Simulation
Serial	Serialization of Java objects from the JBoss benchmark.
XML Transform	XML style sheet processing using `javax.xml.transform`.
XML Validation	XML validation using `javax.xml.validation`.

Table 9.2: Spec 2008 Benchmark Suite

Benchmark	Section	Type	Description
Series	2	Kernel	Fourier Coefficient Analysis
LUFact	2	Kernel	LU Factorization
SOR	2	Kernel	Successive Over-Relaxation
Heap Sort	2	Kernel	Heap Sort Algorithm
Crypt	2	Kernel	IDEA Encryption
FFT	2	Kernel	Fast Fourier Fransform
Sparse	2	Kernel	Sparse Matrix Multiplication
Search	3	Large	Alpha-Beta Pruned Search
Euler	3	Large	Computational Fluid Dynamics
MolDyn	3	Large	Molecular dynamics simulation
Monte Carlo	3	Large	Monte Carlo Simulation
Ray Tracer	3	Large	3D Ray Tracer

Table 9.1: Java Grande Benchmark Suite

Spec 2008 Benchmark Suite We have chosen 20 benchmarks from the Spec 2008 benchmark suite. The benchmark suite contains two other benchmarks that were excluded due to regressions in Maxine or Maxpath. Table 9.2 describes each benchmark in detail.

Mode	Methods	Description
Antlr	3,517	A parser generator and translator generator.
Bloat	5,231	Performs a number of optimizations and analysis on Java bytecode files.
Eclipse	12,450	Executes some of the (non-gui) JDT performance tests for the Eclipse IDE.
Hsqldb	5,970	A SQL database engine written in Java.
Jython	9,443	A Python interpreter written in Java that interprets the pybench Python benchmark.
LUIndex	3,118	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.
Pmd	6,163	Analyzes a set of Java classes for a range of source code problems.

Table 9.3: DaCapo Benchmark Suite — Method count metrics referenced from [BGH+06].

DaCapo Benchmark Suite We have chosen 7 benchmarks from the DaCapo 2006 benchmark suite. The benchmark suite contains four other benchmarks (Fop, Chart, LU Search, Xalan) that were excluded from our evaluation due to regressions in Maxine or Maxpath. The DaCapo benchmark suite focuses on more general purpose object-oriented Java application and includes benchmarks that are much more complex than Java Grande and Spec 2008. For instance, the DaCapo Eclipse benchmark is nearly 10× larger than the Compiler Compiler Spec 2008 benchmark which is by far the largest in its suite.

The DaCapo benchmark was especially chosen in order to test the worst case for Maxpath. An analysis [BGH+06] on the DaCapo benchmarks suggests that it contains significantly less "hot" code than Spec 2008; 9.1% vs 18.0%. For instance, the Eclipse benchmarks has ≈ 4 thousand methods compiled but only 14 of them are considered "hot" (≈ 0.4 %). Moreover, on average DaCapo has 6 × more L1 I-Cache misses than the SPEC benchmarks suggesting much higher dynamic code complexity. This is particularly challenging for Maxpath because it will never have the opportunity recuperate its profiling overhead from optimizations to frequently executed code. Table 9.3 describes each benchmark in detail.

9.2 Execution Modes

Maxine is a sophisticated software system, containing over half a million lines of code with over 200 configurable runtime options. In order to measure our trace-based compiler (Maxpath) within Maxine we compare four execution modes, two Maxine modes and two Maxpath modes (summarized in table 9.5 on page 112):

- T1X : In this mode, Maxine only uses the T1X baseline compiler with no profile guided optimizations and recompilation. However, even in this mode, Maxine may sometimes use the C1X compiler to compile methods that cannot be compiled with the baseline compiler; these include methods that are usually found as part of the JDK that make use of unsafe operations provided by Maxine. On average, across all benchmarks, the number of C1X methods compiled in relation to the total number of T1X methods

compiled is insignificant: $\frac{\text{Unsafe C1X Methods Compiled}}{\text{Total T1X Methods Compiled}} \approx 0.0069$, less than 1%. Therefore, in our evaluation, we ignore this behavior and consider that the T1X compiler is the sole compiler. Unless otherwise specified, T1X is always used as the baseline in tables and charts. [2]

- C1X : In this mode, Maxine uses the T1X baseline compiler with C1X recompilation. T1X instruments all methods with profiling instrumentation. It counts the number of method invocations as well as the number of taken backward branches in each method. If either of these exceed the recompilation threshold (5000 by default) it then recompiles the method using the C1X compiler and patches all callers and VTables to point to the new recompiled method. Maxine does not yet implement on stack replacement (OSR); once a method is recompiled, it can only be entered through the standard entry point at the next invocation and never hot-swapped during execution. This leads undesired behavior that a long running method which is only executed once is never executed in optimized code because it never gets the opportunity to be executed. In contrast, the trace regions are activated as soon as they are compiled, at the next loop iteration.

  Across all configurations, the C1X compiler operates at Optimization Level 3 (all optimizations) with default options, including method inlining. At the time of this writing, speculative optimizations based on class hierarchy analysis (CHA) and deoptimization support are implemented but not completely functional and are disabled. In the future, enabling them would benefit both the C1X compiler and the trace-based compiler that is built on top of it, since nothing prevents trace regions from being deoptimized the same way C1X methods are.

- MTX (T1X + Maxpath): In this mode, Maxine uses the T1X baseline compiler with trace-based compilation. The default configuration of Maxpath is shown in table 9.4 on the following page. Maxpath uses two thresholds to activate trace recording. Maxine uses a fast low-level mechanism to detect frequently executed loops by allocating a variable local to the method code object for each anchor that starts off at the Ignore Threshold (10000 by default) and counts down to zero. Once this threshold is met, Maxpath relies on a the higher-level (and slower) Tracer (section 5.3.2 on page 55) to perform trace profiling which in turn uses the Threshold (100 by default). This two level threshold system is necessary to relieve the stress on the Tracer by filtering out a large number of `visitAnchor` messages originating from many, yet individually insignificant trace anchors. The additional Threshold is subsequently used as a backoff threshold (see backoff threshold in section 5.3.3 on page 59).

- MCX (C1X + Maxpath): In this mode, Maxine uses the T1X baseline compiler with trace-based compilation similar to the MTX mode. However it also uses the C1X compiler to recompile methods that have exceeded an invocation frequency. If we allow the C1X compiler to recompile methods based on backward branches, then it would never let Maxpath run and would be almost equivalent to the C1X mode. In this mode, we only use the C1X compiler for methods that contain code that is frequently executed yet not part of a loop and would not otherwise be compiled by the trace compiler.

[2] The baseline compiler is discussed in more detail in section 5.2.3 on page 47.

Maxpath Option	Default	Description
Ignore Threshold	10000	Number of backward branches to ignore before sending `visitAnchor` messages to the Tracer.
Threshold	100	Number of `visitAnchor` messages before attempting a region extension.
Disable Threshold	500	Number of additional `visitAnchor` messages before the Anchor is disabled.
Max Blocks in Extension	16	Maximum number of blocks in region extension attempt; this controls the size of recorded traces.
Max Extension Attempts	4	Maximum number of extension attempts before the Anchor is disabled.
Max Partial Inline Level	1	Maximum depth of partial inlining.
Min Extensions	4	Minimum number of extension attempts before the trace region is compiled.
Max Unroll Count	1	Maximum number of iterations to unroll (1 = no unrolling).
Max Sub Anchors	4	Maximum number of inner loops to unroll.

Table 9.4: Default Maxpath Options

Mode	Description
T1X	Single mode execution using the T1X baseline compiler with no C1X recompilation.
C1X	Mixed mode execution using the T1X baseline compiler with C1X recompilation. Method invocation count and backward branches trigger C1X recompilation.
MTX	Mixed mode execution using the T1X baseline compiler with Trace-Based Compilation.
MCX	Mixed mode execution using the T1X baseline compiler with Trace-Based Compilation and C1X recompilation. Only method invocation count triggers C1X recompilation; backward branches are handled exclusively by the Trace-Based compiler.

Table 9.5: Execution modes.

9.3 Performance

We compare the results of each of the benchmark suites running under both of the Maxpath trace-compilation modes (MTX, MCX) against each of the Maxine modes (T1X, C1X).

9.3.1 Java Grande

For Java Grande we observe a 2.0184× improvement for MTX and a 2.7977× improvement for MCX over the baseline T1X (shown in figure 9.2 on page 115 (top)). When using the C1X mode as the baseline we observe a 1.2259× and 1.5002× improvement for MTX and MCX respectively (bottom). On average, Maxpath (MCX) is ≈ 50% faster than the best configuration of Maxine. It performs best on SOR and worst on MolDyn. Speedups are

plotted in figure 9.5a on page 118 with a T1X baseline.

9.3.2 Spec 2008

For Spec 2008 (figure 9.3 on page 116) we observe a 2.3336× and 3.7918× speedup over the T1X baseline, and a 1.2370× and 1.5691× speedup over the C1X baseline on average. The biggest performance difference appears in the Scimark suite of benchmarks: FFT, LU, SOR, Sparse and Monte Carlo. A possible explanation for this is that unlike Maxpath, the C1X mode fails to execute hot program regions in optimized code due to its inability to perform on stack replacement.

Particularly interesting are the results for the Compress benchmark, where Maxpath is nearly 4× slower than C1X. The benchmark compresses and decompresses of a stream of bytes several times. The primary reason for the slowdown is in the `decompress()` method shown in figure 9.1 on the following page. The method is called less than 1000 times, but each individual invocation takes a long period of time. The trace compiler detects and compiles trace regions for the two inner loops at lines 30 and 39, but these loops are quite small and don't contribute enough to the total execution time of the method. Compiling the outer loop is more important, but doesn't happen because the tracer cannot unroll the execution of the two inner loops (an example CFG discussed previously in section 3.2.2 on page 19). In such cases, the C1X compiler would normally take over and compile the method, but since only method invocation count triggers recompilation in the MCX mode and the invocation count never exeeds the recompilation threshold (5000 by default) the method is never compiled with the C1X compiler. If we tell Maxpath to ignore the `decompress()` method, the performance of the benchmark in MCX mode increases significantly, from 8.37 ops/m to 22.32 ops/m, a nearly 2.6666× faster.

In theory, a heuristic could be built to detect such cases and disable trace compilation, but we have not explored this research area. Moreover, the nested trace trees approach described in section 3.2.2 on page 19 could be used to capture this type of control flow, but we have not implemented this technique in Maxpath.

9.3.3 DaCapo

On the much larger DaCapo benchmarks we observe a 1.0628× and 1.7825× speedup over the T1X baseline, and a 0.5725× and 0.9226× speedup over the C1X baseline on average. Speedups across all benchmarks are plotted in figure 9.5 on page 118.

9.3.4 Hotspot Comparison

Maxine is a research JVM and is nowhere nearly as tuned and mature as the Hotspot JVM. On average, Maxine is ≈ 50% slower than Hotspot across all benchmark suites (shown in figure 9.6 on page 119).

```
public void decompress() {
    int code, oldcode, incode;

    int finchar = oldcode = getCode();
    if (oldcode == -1) {/* EOF already? */
        return; /* Get out of here */
    }
    output.writeByte((byte) finchar); /* first code must be 8 bits = byte */

    while ((code = getCode()) > -1) {
        if ((code == Compress.CLEAR) && (blockCompress != 0)) {
            tabPrefix.clear(256);
            clearFlag = 1;
            freeEntry = Compress.FIRST - 1;
            if ((code = getCode()) == -1) /* O, untimely death! */
                break;
        }
        incode = code;
        /*
         * Special case for KwKwK string.
         */
        if (code >= freeEntry) {
            deStack.push((byte) finchar);
            code = oldcode;
        }

        /*
         * Generate output characters in reverse order
         */
        while (code >= 256) {
            deStack.push(tabSuffix.of(code));
            code = tabPrefix.of(code);
        }
        deStack.push((byte) (finchar = tabSuffix.of(code)));

        /*
         * And put them out in forward order
         */
        do {
            output.writeByte(deStack.pop());
        } while (!deStack.isEmpty());

        /*
         * Generate the new entry.
         */
        if ((code = freeEntry) < maxMaxCode) {
            tabPrefix.set(code, oldcode);
            tabSuffix.set(code, (byte) finchar);
            freeEntry = code + 1;
        }
        /*
         * Remember previous code.
         */
        oldcode = incode;
    }
}
```

Figure 9.1: Hot `decompress()` method in the Compress benchmark.

Java Grande — T1X vs. MTX and MCX

Benchmark	T1X (s)	MTX (s)	× MTX	MCX (s)	× MCX
LUFact	8.91	1.80	× 4.94	1.80	× 4.94
Heap Sort	8.31	8.11	× 1.02	2.90	× 2.86
Crypt	9.31	4.11	× 2.27	4.11	× 2.27
FFT	41.72	18.61	× 2.24	17.81	× 2.34
SOR	9.21	2.90	× 3.17	2.80	× 3.28
Sparse Matmult	4.11	1.70	× 2.41	1.60	× 2.56
Series	70.93	62.83	× 1.13	55.03	× 1.29
Euler	41.52	14.81	× 2.80	14.41	× 2.88
MolDyn	10.61	10.61	× 1.00	3.10	× 3.42
Monte Carlo	9.21	7.41	× 1.24	6.71	× 1.37
Ray Tracer	17.21	17.61	× 0.98	5.51	× 3.13
Search	10.71	10.51	× 1.02	3.31	× 3.24
Average			× 2.02		× 2.80

Java Grande — C1X vs. MTX and MCX

Benchmark	C1X (s)	MTX (s)	× MTX	MCX (s)	× MCX
LUFact	1.90	1.80	× 1.06	1.80	× 1.06
Heap Sort	3.30	8.11	× 0.41	2.90	× 1.14
Crypt	9.41	4.11	× 2.29	4.11	× 2.29
FFT	41.12	18.61	× 2.21	17.81	× 2.31
SOR	9.21	2.90	× 3.17	2.80	× 3.28
Sparse Matmult	4.11	1.70	× 2.41	1.60	× 2.56
Series	55.03	62.83	× 0.88	55.03	× 1.00
Euler	7.61	14.81	× 0.51	14.41	× 0.53
MolDyn	2.61	10.61	× 0.25	3.10	× 0.84
Monte Carlo	6.71	7.41	× 0.91	6.71	× 1.00
Ray Tracer	5.51	17.61	× 0.31	5.51	× 1.00
Search	3.30	10.51	× 0.31	3.31	× 1.00
Average			× 1.23		× 1.50

Figure 9.2: Java Grande execution times, and speedup factors (×) of MTX and MCX relative to T1X (top) and C1X (bottom).

Spec 2008 — T1X vs. MTX and MCX

Benchmark	T1X (score)	MTX (score)	× MTX	MCX (score)	× MCX
Compiler Compiler	20.94	23.01	× 1.10	52.25	× 2.50
Compiler Sunflow	7.78	8.09	× 1.04	17.74	× 2.28
Compress	4.8	8.14	× 1.70	8.37	× 1.74
Crypto AES	1.2	2.77	× 2.31	8.14	× 6.78
Crypto RSA	5.62	15.93	× 2.83	32.48	× 5.78
Crypto Signverify	4.19	14.39	× 3.43	29.57	× 7.06
Derby	6.69	6.83	× 1.02	19.4	× 2.90
Mpegaudio	2.21	3.32	× 1.50	13.07	× 5.91
Scimark FFT Large	3.52	8.62	× 2.45	8.73	× 2.48
Scimark LU Large	0.45	2.79	× 6.20	2.79	× 6.20
Scimark SOR Large	1.49	5.88	× 3.95	5.89	× 3.95
Scimark Sparse Large	1.41	6.05	× 4.29	6.05	× 4.29
Scimark Monte Carlo	1.59	1.56	× 0.98	1.61	× 1.01
Serial	4.74	5.42	× 1.14	7.07	× 1.49
XML Validation	10.6	11.22	× 1.06	26.45	× 2.50
Average			× 2.33		× 3.79

Spec 2008 — C1X vs. MTX and MCX

Benchmark	C1X (score)	MTX (score)	× MTX	MCX (score)	× MCX
Compiler Compiler	50.59	23.01	× 0.45	52.25	× 1.03
Compiler Sunflow	18.62	8.09	× 0.43	17.74	× 0.95
Compress	29.26	8.14	× 0.28	8.37	× 0.29
Crypto AES	8.53	2.77	× 0.32	8.14	× 0.95
Crypto RSA	32.3	15.93	× 0.49	32.48	× 1.01
Crypto Signverify	29.21	14.39	× 0.49	29.57	× 1.01
Derby	20.62	6.83	× 0.33	19.4	× 0.94
Mpegaudio	13.09	3.32	× 0.25	13.07	× 1.00
Scimark FFT Large	8.55	8.62	× 1.01	8.73	× 1.02
Scimark LU Large	0.76	2.79	× 3.67	2.79	× 3.67
Scimark SOR Large	1.46	5.88	× 4.03	5.89	× 4.03
Scimark Sparse Large	1.31	6.05	× 4.62	6.05	× 4.62
Scimark Monte Carlo	1.61	1.56	× 0.97	1.61	× 1.00
Serial	6.99	5.42	× 0.78	7.07	× 1.01
XML Validation	26.54	11.22	× 0.42	26.45	× 1.00
Average			× 1.24		× 1.57

Figure 9.3: Spec 2008 execution times, and speedup factors (×) of MTX and MCX relative to T1X (top) and C1X (bottom).

Dacapo — T1X vs. MTX and MCX

Benchmark	T1X (s)	MTX (s)	× MTX	MCX (s)	× MCX
Bloat	23.27	23.62	× 0.99	11.63	× 2.00
Eclipse	110.72	61.16	× 1.81	53.95	× 2.05
Hsqldb	9.28	9.73	× 0.95	6.13	× 1.51
Jython	13.64	23.82	× 0.57	18.50	× 0.74
luindex	18.43	18.05	× 1.02	6.36	× 2.90
Pmd	14.49	14.01	× 1.03	9.69	× 1.50
Average			× 1.06		× 1.78

Dacapo — C1X vs. MTX and MCX

Benchmark	C1X (s)	MTX (s)	× MTX	MCX (s)	× MCX
Bloat	11.58	23.62	× 0.49	11.63	× 1.00
Eclipse	32.89	61.16	× 0.54	53.95	× 0.61
Hsqldb	6.02	9.73	× 0.62	6.13	× 0.98
Jython	17.86	23.82	× 0.75	18.50	× 0.97
luindex	6.25	18.05	× 0.35	6.36	× 0.98
Pmd	9.69	14.01	× 0.69	9.69	× 1.00
Average			× 0.57		× 0.92

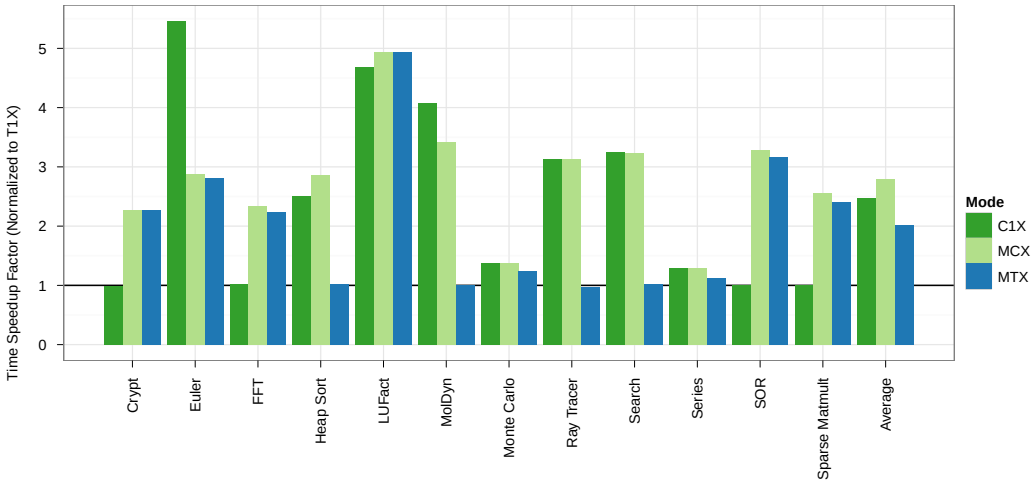
Figure 9.4: DaCapo execution times, and speedup factors (×) of MTX and MCX relative to T1X (top) and C1X (bottom).

9.4 Memory Consumption

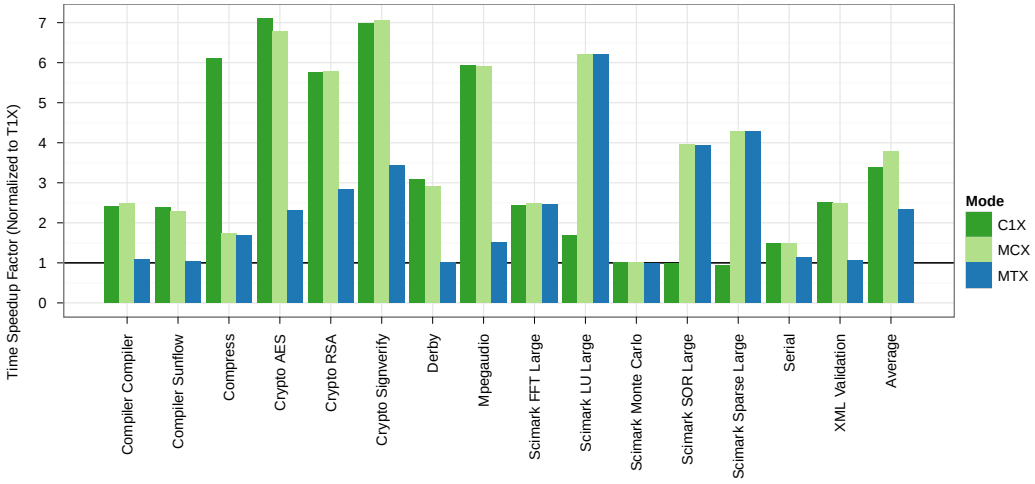
9.4.1 Profiling Instrumentation

In order to measure the *static* code size overhead of profiling instrumentation (figure 9.7 on page 120), we used the T1X compiler to compile ahead-of-time a large set Java methods (≈ 60,000 methods from the JDK and the Maxine source base) under three configurations (table 9.6 on page 120). We have observed that PIM instrumentation overhead is very small ≈ 0.52 % up to 31,580.77K from 31,417.94K. TIM overhead is significantly larger, ≈ 145.77 % up to 77,215.27K from 31,417.94K, nearly × 2.46 larger. This is prohibitively large, however is not as bad as it seems because TIM code is only seldom generated, on demand, whenever needed.

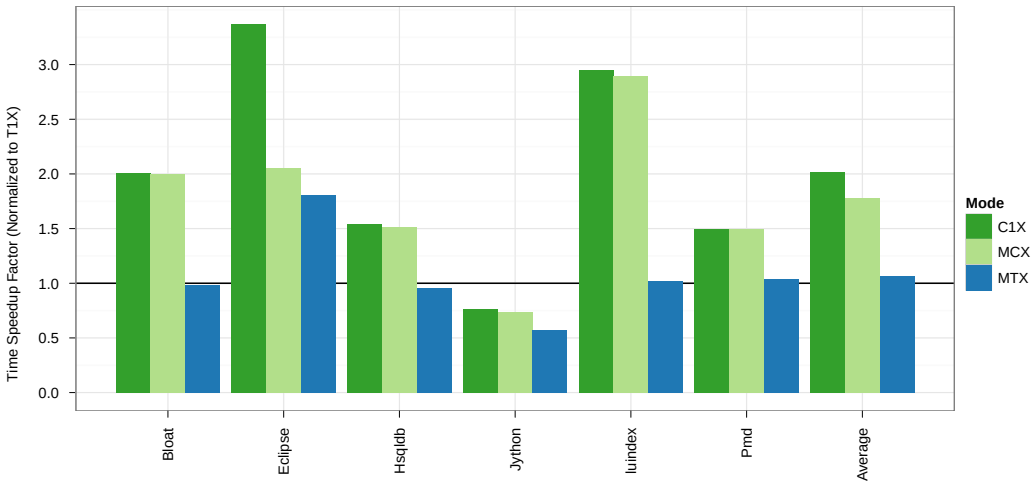
In order to measure the *actual* dynamic overhead of these instrumentation modes (in terms of code size), we measured the total code size of T1X generated methods for a set of benchmarks under three configurations: (1) with no instrumentation at all (NIM); (2) with profiling instrumentation (PIM), with a sufficiently large Ignore Threshold so that no trace would ever be recorded (thus no need for TIM code); (3) and lastly, under normal execution (PIM + TIM) where PIM code is generated for profiling and additional TIM code is generated on demand for trace recording. Figure 9.8 on page 122 shows the generated code size for each of the three configurations (NIM, PIM and PIM + TIM) as well as their respective increase in code size over NIM code (PIM % and PIM + TIM %). The results indicate that PIM overhead is 2.93% for Java Grande, 1.65% for Spec 2008 and 2.02% for DaCapo. More



(a) Java Grande Benchmark Speedups

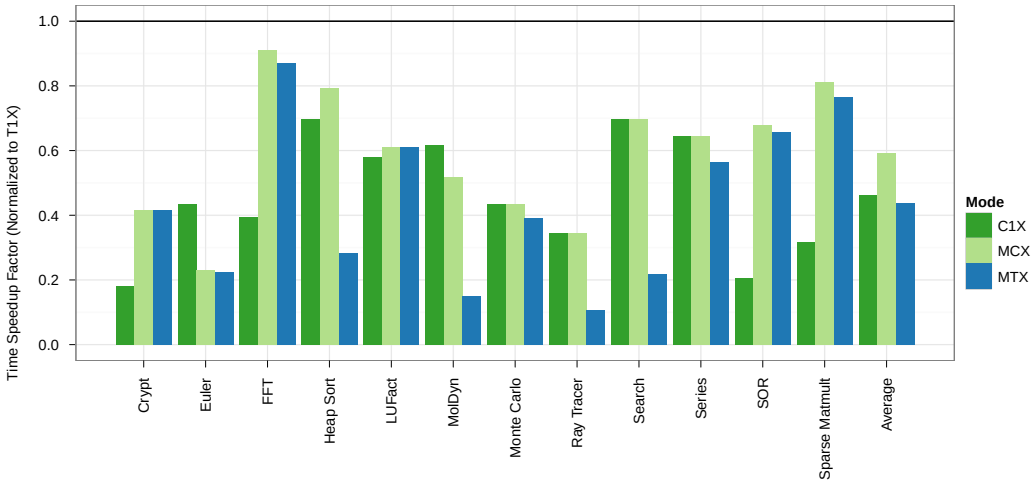


(b) Spec 2008 Benchmark Speedups

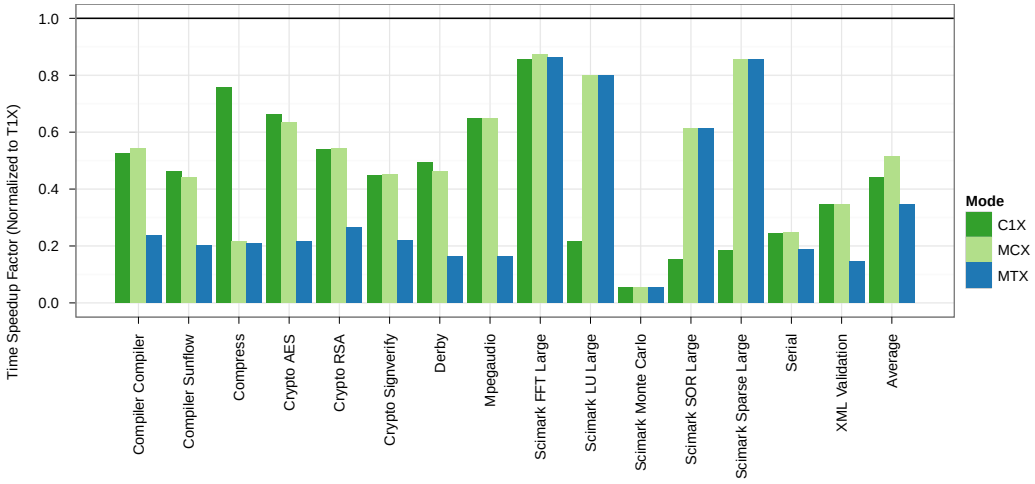


(c) DaCapo Benchmark Speedups

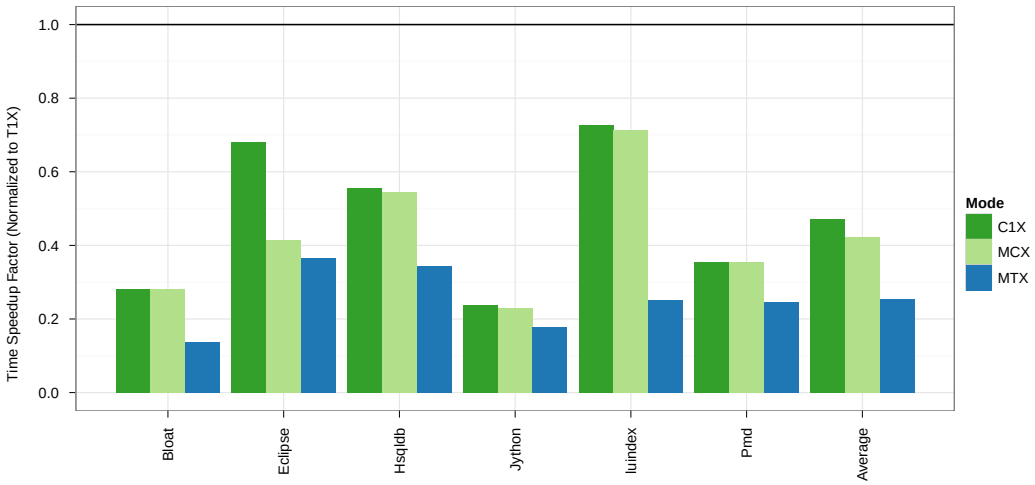
Figure 9.5: Relative speedups of C1X, MTX and MCX compared to T1X baseline.



(a) Java Grande Benchmark — Maxine vs Hotspot



(b) Spec 2008 Benchmark — Maxine vs Hotspot



(c) DaCapo Benchmark — Maxine vs Hotspot

Figure 9.6: Relative speedups of C1X, MTX and MCX compared to Hotspot baseline.

NIM (bytes)	PIM (bytes)	PIM %	TIM (bytes)	TIM %
31,417.94K	31,580.77K	0.52 %	77,215.27K	145.77 %

Figure 9.7: Static Profiling Overhead

Configuration	Description
NIM	*No Instrumentation*: Methods were compiled without any instrumentation.
PIM	*Profile Instrumentation*: Methods were compiled with anchor instrumentation at each loop header.
TIM	*Trace Instrumentation*: Methods were compiled with anchor instrumentation at each loop header and with tracing instrumentation at each bytecode. Tracing instrumentation significantly increases the size of the compiled code, by a factor of $\approx 2.45\times$.

Table 9.6: Instrumentation types.

importantly, the overhead of PIM + TIM is only 21.06%, 13.80% and 27.42% for the three benchmark suites, much smaller than $\times$ 2.46 for the static case. Because methods compiled with TIM are only used during trace recording and generated on demand, this overhead is amortized.

9.4.2 Code Size

Figure 9.10 on page 124 tabulates the amount of code generated for each of the three benchmark suites. We observe a significant reduction in code size when using Maxpath (MTX and MCX). On average, Maxpath produces 0.88×, 0.29×, 0.14× as much code as Maxine under MTX ($\frac{\text{MTX Bytes Compiled}}{\text{C1X Bytes Compiled}}$) and 1.24×, 0.78×, and 0.69× for MCX ($\frac{\text{MCX Bytes Compiled}}{\text{C1X Bytes Compiled}}$). This is indicative of the fact that we compile a much smaller subset of the code than the C1X compiler (C1X mode), which also implies that compilation time is reduced correspondingly.

On average, Maxpath (MTX) outperforms Maxine's method based compilation strategy (C1X) on Java Grande and Spec 2008 by $\approx$ 23% and loses on the DaCapo benchmarks by a factor of 0.57×. However, it also produces nearly half as much code (MCX mode). We plot the relationship between execution speed and code size for MTX and MCX ($\frac{\text{MTX Speedup over C1X}}{\text{MTX Code Size / C1X Code Size}}$ and $\frac{\text{MCX Speedup over C1X}}{\text{MCX Code Size / C1X Code Size}}$) in figure 9.9 on page 123 and compare it to the C1X baseline.

The results indicates that Maxpath is much better at choosing what to compile than Maxine, but ultimately suffers from not compiling enough. For example, in the Eclipse benchmark, MTX is nearly twice as slow as C1X (0.54 ×) however, C1X compiles 2,190 methods, while MTX only compiles 37 methods and 196 regions, resulting in significant reduction in code size (0.07 ×).

9.5 Tracer Recording Metrics

Figure 9.11 on page 125 and figure 9.12 on page 126 tabulate the following trace recording metrics:

- Visit Anchor : Number of `visitAnchor` messages sent to the tracer.
- Visit Sub Anchor : Number of `visitAnchor` messages sent to the tracer while recording a trace. This indicates the number of inner loop headers that are reached during trace recording.
- Visit Blocks & Bytecode : Number of `visitBlock` and `visitBytecode` messages sent to the tracer.
- Visit Methods : Number of method call sites visited during trace recording.
- Methods Inlined & Partially Inlined : Number of statically inlined methods and partially inlined methods. The number of partially inlined methods is higher on most benchmarks.
- Region & Method Live-In Arguments : Average number of live-in arguments to compiled regions and methods. On average, regions have at least twice as many live-in arguments flowing into them as methods. This negatively impacts the calling convention overhead for regions, since more values have to be marshaled in and out of compiled code.

9.5.1 Trace Region Metrics

Figure 9.13 on page 127 presents several interesting metrics about trace regions:

- Anchors : Number of Anchors statically inserted into PIM code.
- Traces : Number of recorded traces.
- Regions : Number of constructed regions.
- Transfers : Total number of transfers in the constructed regions.
- $\frac{\text{Region}}{\text{Anchor}}$: Percentage of anchors that wound up to have regions constructed for them.
- $\frac{\text{Traces}}{\text{Region}}$: Average number of traces per region.
- $\frac{\text{Transfers}}{\text{Region}}$: Average number of transfers per region. This indicates that trace regions have relatively low number of transfers, as compared to previous trace-based compilation techniques [BBF+10].

Java Grande — NIM vs. PIM and PIM+TIM

Benchmark	Methods	NIM (bytes)	PIM (bytes)	PIM %	PIM+TIM (bytes)	PIM+TIM %
LUFact	401	301.45K	310.97K	3.16 %	343.93K	14.09 %
Heap Sort	402	282.57K	290.32K	2.74 %	303.98K	7.58 %
Crypt	404	299.18K	307.42K	2.76 %	334.59K	11.84 %
FFT	406	292.33K	300.67K	2.85 %	327.21K	11.93 %
SOR	402	283.92K	291.87K	2.80 %	302.12K	6.41 %
Sparse Matmult	406	286.25K	294.00K	2.71 %	305.87K	6.85 %
Series	401	285.82K	293.46K	2.68 %	302.75K	5.93 %
Euler	429	528.49K	541.37K	2.44 %	1,065.84K	101.68 %
MolDyn	409	319.29K	329.10K	3.07 %	358.67K	12.33 %
Monte Carlo	510	359.03K	369.15K	2.82 %	490.17K	36.53 %
Ray Tracer	442	314.60K	328.12K	4.30 %	366.90K	16.63 %
Search	410	326.50K	335.72K	2.83 %	395.00K	20.98 %
Average				2.93 %		21.06 %

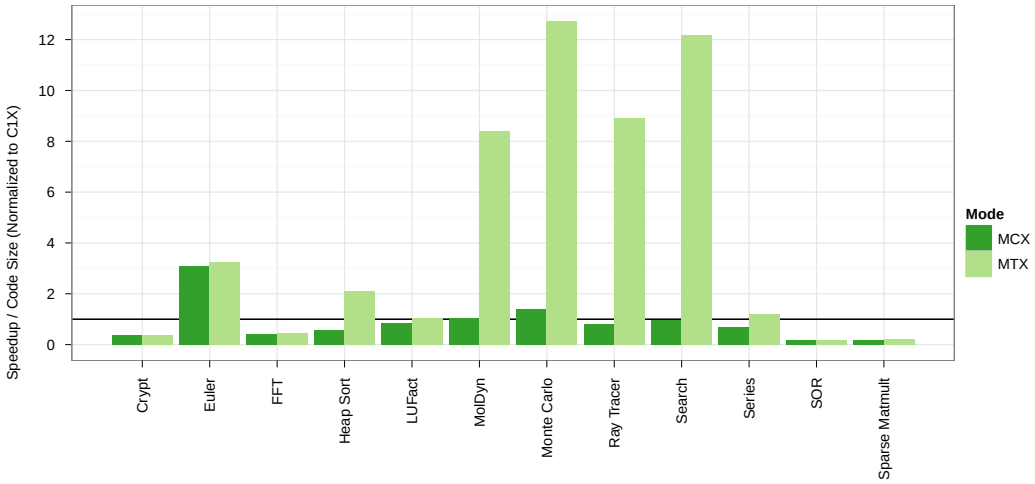
Spec 2008 — NIM vs. PIM and PIM+TIM

Benchmark	Methods	NIM (bytes)	PIM (bytes)	PIM %	PIM+TIM (bytes)	PIM+TIM %
Compiler Compiler	4.50K	5,708.53K	5,806.54K	1.72 %	7,616.57K	33.42 %
Compiler Sunflow	4.37K	5,624.88K	5,721.21K	1.71 %	7,515.71K	33.62 %
Compress	1.78K	2,403.20K	2,439.46K	1.51 %	2,559.55K	6.51 %
Crypto AES	2.39K	3,424.15K	3,474.42K	1.47 %	3,704.80K	8.20 %
Crypto RSA	2.34K	2,931.37K	2,980.07K	1.66 %	3,345.21K	14.12 %
Crypto Signverify	2.10K	2,731.28K	2,778.15K	1.72 %	3,139.04K	14.93 %
Derby	7.68K	8,741.52K	8,886.75K	1.66 %	9,865.27K	12.86 %
Mpegaudio	1.94K	4,655.55K	4,703.33K	1.03 %	5,609.59K	20.49 %
Scimark FFT Large	1.77K	2,410.97K	2,448.72K	1.57 %	2,543.60K	5.50 %
Scimark LU Large	1.77K	2,409.76K	2,459.95K	2.08 %	2,547.08K	5.70 %
Scimark SOR Large	1.75K	2,378.17K	2,413.63K	1.49 %	2,490.75K	4.73 %
Scimark Sparse Large	1.75K	2,376.19K	2,417.79K	1.75 %	2,493.42K	4.93 %
Scimark Monte Carlo	1.74K	2,369.87K	2,404.95K	1.48 %	2,475.20K	4.44 %
Serial	2.21K	2,783.46K	2,847.53K	2.30 %	3,114.22K	11.88 %
XML Validation	3.83K	5,269.43K	5,354.93K	1.62 %	6,618.47K	25.60 %
Average				1.65 %		13.80 %

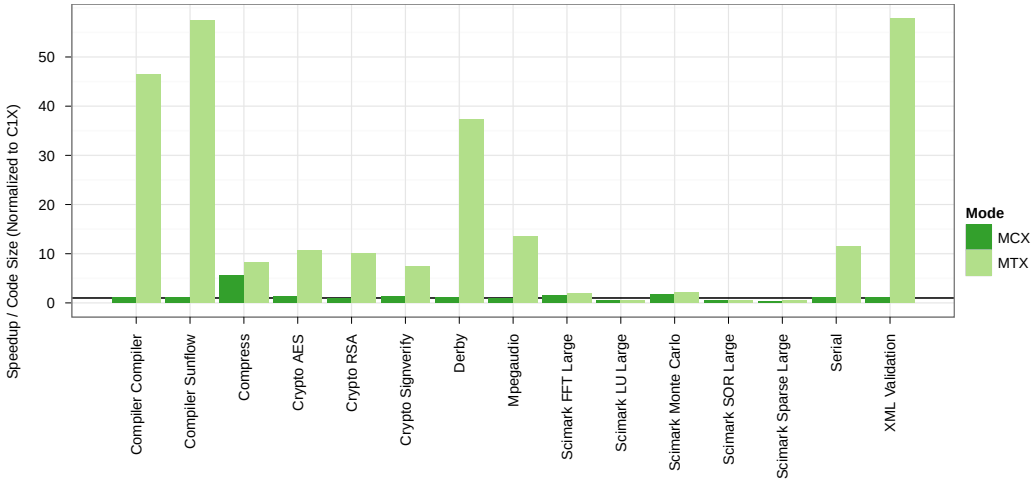
DaCapo 2006 — NIM vs. PIM and PIM+TIM

Benchmark	Methods	NIM (bytes)	PIM (bytes)	PIM %	PIM+TIM (bytes)	PIM+TIM %
Bloat	3.25K	3,451.44K	3,540.22K	2.57 %	4,392.23K	27.26 %
Eclipse	11.69K	12,734.61K	13,010.16K	2.16 %	16,687.67K	31.04 %
Hsqldb	2.74K	2,989.39K	3,053.08K	2.13 %	3,504.75K	17.24 %
Jython	4.77K	8,736.72K	8,827.34K	1.04 %	12,501.37K	43.09 %
luindex	2.20K	2,054.52K	2,101.65K	2.29 %	2,625.73K	27.80 %
Pmd	3.81K	3,443.75K	3,510.92K	1.95 %	4,067.18K	18.10 %
Average				2.02 %		27.42 %

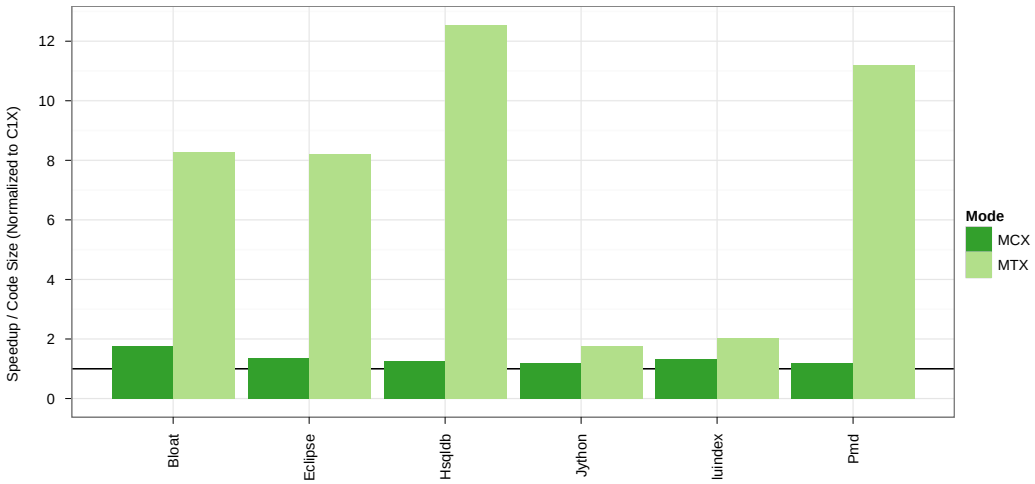
Figure 9.8: Dynamically computed T1X code size for three instrumentation modes: NIM, PIM, PIM+TIM



(a) Java Grande Benchmark Speedup / Code Size



(b) Spec 2008 Benchmark Speedup / Code Size



(c) DaCapo Benchmark Speedup / Code Size

Figure 9.9: Speedup relative to code size of MTX and MCX compared to C1X baseline.

Java Grande — C1X Generated Code — C1X vs. MTX and MCX

	C1X		MTX				MCX			
Benchmark	meth	bytes	meth	reg	bytes	× bytes	meth	reg	bytes	× bytes
LUFact	7	2.84K	2	7	2.61K	× 0.92	4	7	3.22K	× 1.13
Heap Sort	4	1.29K	2	4	1.50K	× 1.17	4	4	1.95K	× 1.52
Crypt	3	2.67K	2	3	3.11K	× 1.16	2	3	3.11K	× 1.16
FFT	5	2.75K	2	7	2.78K	× 1.01	3	7	2.90K	× 1.06
SOR	3	985	2	3	1.55K	× 1.61	3	3	1.67K	× 1.74
Sparse Matmult	5	1.05K	2	4	2.03K	× 1.93	5	4	2.24K	× 2.13
Series	5	1.56K	2	2	1.49K	× 0.96	6	2	2.25K	× 1.44
Euler	24	62.02K	3	20	36.98K	× 0.60	18	20	38.08K	× 0.61
MolDyn	9	3.14K	2	4	1.53K	× 0.49	8	4	3.56K	× 1.13
Monte Carlo	87	38.36K	3	7	3.33K	× 0.09	90	7	27.73K	× 0.72
Ray Tracer	22	5.98K	2	2	2.15K	× 0.36	23	2	7.36K	× 1.23
Search	11	6.74K	2	3	1.76K	× 0.26	11	2	7.10K	× 1.05
Average						× 0.88				× 1.24

Spec 2008 — C1X Generated Code — C1X vs. MTX and MCX

	C1X		MTX				MCX			
Benchmark	meth	bytes	meth	reg	bytes	× bytes	meth	reg	bytes	× bytes
Compiler Compiler	1.66K	910.16K	22	103	43.03K	× 0.05	1.61K	97	822.56K	× 0.90
Compiler Sunflow	1.58K	893.96K	22	92	35.80K	× 0.04	1.51K	87	808.51K	× 0.90
Compress	57	18.78K	15	17	8.11K	× 0.43	41	16	11.83K	× 0.63
Crypto AES	101	72.16K	31	26	20.94K	× 0.29	78	26	59.06K	× 0.82
Crypto RSA	263	143.95K	31	53	29.10K	× 0.20	255	50	141.95K	× 0.99
Crypto Signverify	155	83.54K	21	47	23.14K	× 0.28	132	46	61.97K	× 0.74
Derby	1.18K	493.64K	41	59	40.02K	× 0.08	1.15K	59	442.51K	× 0.90
Mpegaudio	105	159.77K	15	64	46.67K	× 0.29	94	64	165.79K	× 1.04
Scimark FFT Large	41	16.40K	15	16	8.65K	× 0.53	27	16	10.16K	× 0.62
Scimark LU Large	42	17.34K	15	18	8.51K	× 0.49	27	18	10.02K	× 0.58
Scimark SOR Large	38	14.80K	15	12	7.34K	× 0.50	27	12	8.84K	× 0.60
Scimark Sparse Large	37	15.58K	15	14	7.80K	× 0.50	26	14	9.25K	× 0.59
Scimark Monte Carlo	36	14.10K	15	11	6.93K	× 0.49	26	11	8.38K	× 0.59
Serial	276	141.86K	37	23	15.90K	× 0.11	265	24	128.76K	× 0.91
XML Validation	836	477.28K	15	48	19.51K	× 0.04	827	45	447.85K	× 0.94
Average						× 0.29				× 0.78

DaCapo 2006 — C1X Generated Code — C1X vs. MTX and MCX

	C1X		MTX				MCX			
Benchmark	meth	bytes	meth	reg	bytes	× bytes	meth	reg	bytes	× bytes
Bloat	547	599.45K	19	67	35.51K	× 0.06	494	64	337.20K	× 0.56
Eclipse	2.19K	1,442.93K	37	196	94.63K	× 0.07	1.67K	197	643.55K	× 0.45
Hsqldb	415	261.00K	22	18	12.88K	× 0.05	387	18	203.18K	× 0.78
Jython	790	1,116.45K	21	97	476.37K	× 0.43	726	95	891.88K	× 0.80
luindex	301	175.08K	26	33	29.61K	× 0.17	289	34	130.15K	× 0.74
Pmd	608	264.71K	19	31	16.37K	× 0.06	590	30	222.34K	× 0.84
Average						× 0.14				× 0.69

Figure 9.10: Code generated by the C1X compiler under three execution modes: C1X, MTX, MCX. Legend: meth number of methods compiled by the C1X compiler, regs number of regions compiled by the C1X compiler, bytes number of code bytes produced by the C1X compiler.

Java Grande — Trace Recording Metrics

Benchmark	Visit Anchors	Visit Sub Anchors	Visit Blocks	Visit Bytecodes	Visit Method	Unwinds	Methods Inlined	Methods Partially Inlined	Region Live-In Arguments	Method Live-In Arguments
LUFact	812	0	183	1.21K	16	7	0	1	× 9.86	× 4.57
Heap Sort	1.06K	0	140	770	24	4	0	0	× 3.50	× 2.25
Crypt	349	1	79	1.32K	0	3	0	0	× 7.00	× 3.00
FFT	1.46K	2	236	4.88K	8	12	0	0	× 7.29	× 1.80
SOR	989	0	150	1.79K	8	8	0	0	× 11.00	× 2.00
Sparse Matmult	464	0	86	688	48	4	0	2	× 5.75	× 1.60
Series	232	0	42	431	26	2	2	1	× 6.00	× 3.40
Euler	13.07K	8	1.97K	133.29K	614	105	6	14	× 11.90	× 1.83
MolDyn	1.06K	0	170	1.39K	32	9	0	0	× 1.00	× 2.22
Monte Carlo	5.69K	44	453	2.83K	102	8	1	4	× 5.14	× 1.66
Ray Tracer	860	0	148	1.78K	87	3	3	3	× 12.50	× 2.73
Search	5.83K	57	468	2.90K	34	6	0	1	× 8.67	× 2.36

Spec 2008 — Trace Recording Metrics

Benchmark	Visit Anchors	Visit Sub Anchors	Visit Blocks	Visit Bytecodes	Visit Method	Unwinds	Methods Inlined	Methods Partially Inlined	Region Live-In Arguments	Method Live-In Arguments
Compiler Compiler	145.07K	399	10.58K	55.84K	5.58K	199	16	53	× 6.02	× 2.38
Compiler Sunflow	139.77K	340	10.05K	52.74K	5.08K	184	17	55	× 5.78	× 2.34
Compress	5.04K	54	701	6.25K	231	24	0	7	× 6.65	× 2.21
Crypto AES	6.15K	13	921	10.31K	151	41	2	3	× 6.77	× 2.31
Crypto RSA	22.59K	71	2.50K	18.83K	511	85	12	13	× 8.06	× 2.03
Crypto Signverify	17.72K	50	2.01K	17.55K	350	73	18	16	× 8.06	× 2.26
Derby	57.79K	85	5.21K	30.79K	1.76K	168	21	42	× 6.39	× 2.22
Mpegaudio	24.13K	122	3.46K	52.27K	374	151	3	7	× 9.19	× 2.36
Scimark FFT Large	3.74K	14	569	8.81K	40	26	0	0	× 8.19	× 2.39
Scimark LU Large	3.97K	12	614	5.75K	48	28	0	1	× 8.78	× 2.45
Scimark SOR Large	3.28K	12	483	5.63K	40	22	0	0	× 8.25	× 2.45
Scimark Sparse Large	2.87K	14	438	4.33K	40	19	0	0	× 8.93	× 2.49
Scimark Monte Carlo	2.52K	12	388	4.06K	48	16	0	0	× 7.55	× 2.42
Serial	14.81K	22	1.58K	10.54K	529	42	1	9	× 6.91	× 2.17
XML Validation	68.03K	224	6.16K	34.61K	1.78K	158	9	33	× 6.48	× 2.11

Figure 9.11: Tracing Visit Metrics for Java Grande and Spec 2008.

DaCapo — Trace Recording Metrics

Benchmark	Visit Anchors	Visit Sub Anchors	Visit Blocks	Visit Bytecodes	Visit Method	Unwinds	Methods Inlined	Methods Partially Inlined	Region Live-In Arguments	Method Live-In Arguments
Bloat	44.88K	324	4.63K	23.07K	3.42K	120	30	0	× 6.87	× 1.78
Eclipse	204.25K	678	17.93K	89.51K	5.08K	403	47	88	× 6.95	× 2.12
Hsqldb	34.39K	111	2.28K	13.75K	1.16K	36	2	0	× 6.78	× 2.04
Jython	62.53K	148	5.67K	307.76K	77.76K	151	3.12K	0	× 5.18	× 2.05
luindex	29.66K	50	2.78K	16.60K	1.54K	82	41	0	× 7.45	× 2.07
Pmd	39.08K	56	2.84K	14.29K	1.23K	70	56	0	× 6.74	× 1.90

Figure 9.12: Tracing Visit Metrics for DaCapo.

Java Grande — Trace Region Metrics

Benchmark	Anchors	Traces	Regions	Transfers	$\frac{\text{Region}}{\text{Anchor}}$	$\frac{\text{Traces}}{\text{Region}}$	$\frac{\text{Transfers}}{\text{Region}}$
LUFact	96	28	7	10	7.29 %	4.00	1.43
Heap Sort	79	16	4	6	5.06 %	4.00	1.50
Crypt	84	12	3	5	3.57 %	4.00	1.67
FFT	85	28	7	7	8.24 %	4.00	1.00
SOR	81	12	3	3	3.70 %	4.00	1.00
Sparse Matmult	79	16	4	6	5.06 %	4.00	1.50
Series	78	8	2	5	2.56 %	4.00	2.50
Euler	136	80	20	28	14.71 %	4.00	1.40
MolDyn	99	16	4	4	4.04 %	4.00	1.00
Monte Carlo	103	46	7	19	6.80 %	6.57	2.71
Ray Tracer	88	11	2	5	2.27 %	5.50	2.50
Search	93	22	3	10	3.23 %	7.33	3.33

Spec 2008 — Trace Region Metrics

Benchmark	Anchors	Traces	Regions	Transfers	$\frac{\text{Region}}{\text{Anchor}}$	$\frac{\text{Traces}}{\text{Region}}$	$\frac{\text{Transfers}}{\text{Region}}$
Compiler Compiler	1009	541	103	262	10.21 %	5.25	2.54
Compiler Sunflow	993	545	92	227	9.26 %	5.92	2.47
Compress	371	73	17	24	4.58 %	4.29	1.41
Crypto AES	521	107	26	33	4.99 %	4.12	1.27
Crypto RSA	520	233	53	100	10.19 %	4.40	1.89
Crypto Signverify	469	195	47	78	10.02 %	4.15	1.66
Derby	1.48K	300	59	139	3.99 %	5.08	2.36
Mpegaudio	489	271	64	93	13.09 %	4.23	1.45
Scimark FFT Large	386	64	16	18	4.15 %	4.00	1.12
Scimark LU Large	393	72	18	22	4.58 %	4.00	1.22
Scimark SOR Large	363	48	12	14	3.31 %	4.00	1.17
Scimark Sparse Large	364	56	14	17	3.85 %	4.00	1.21
Scimark Monte Carlo	359	44	11	13	3.06 %	4.00	1.18
Serial	432	114	23	56	5.32 %	4.96	2.43
XML Validation	877	265	48	171	5.47 %	5.52	3.56

DaCapo — Trace Region Metrics

Benchmark	Anchors	Traces	Regions	Transfers	$\frac{\text{Region}}{\text{Anchor}}$	$\frac{\text{Traces}}{\text{Region}}$	$\frac{\text{Transfers}}{\text{Region}}$
Bloat	838	319	67	127	8.00 %	4.76	1.90
Eclipse	3.18K	1.09K	196	594	6.16 %	5.57	3.03
Hsqldb	655	104	18	31	2.75 %	5.78	1.72
Jython	979	432	97	161	9.91 %	4.45	1.66
luindex	488	179	33	185	6.76 %	5.42	5.61
Pmd	680	147	31	56	4.56 %	4.74	1.81

Figure 9.13: Trace Region Metrics

Chapter 10

Conclusions

In conclusion, we have explored the design and implementation of trace-based compilation in several contexts and have shown how it can be used to improve the performance programs, while at the same time generate significantly less code.

This dissertation makes the following contributions:

- We have provided a historical perspective on trace-based compilation and discussed our early efforts on trace-based compilation in partially meta-circular virtual machines and their influences on modern trace-based compiler implementations such as TraceMonkey in the Mozilla Firefox browser.
- We have shown that trace-based compilation is feasible and beneficial in virtual execution environments without interpreters. We have introduced trace regions as the primary compilation unit and have shown how they can be recorded and grown using dynamic instrumentation.
- We described how a Java virtual machine can be retrofitted to support trace-based compilation with relatively little effort.
- We have extended the trace-oriented compilation paradigm in the direction of more traditional control flow modeling, while retaining its exclusive focus on hot program loops. The result is a very nimble trace-based optimizing compiler in which the SSA form intermediate representation is created truly incrementally in lockstep with execution.
- Finally, we have explored trace-based compilation in the context of dynamically typed languages that are layered on top of statically typed languages such as Java and C#.

10.1 Acknowledgements

Parts of this effort have been sponsored by the National Science Foundation under grants CNS-0615443 and CNS-0627747, as well as by the California MICRO Program and industrial sponsor Sun Microsystems under Project No. 07-127. Further support in the form of generous unrestricted gifts has come from Google and from Mozilla, for which I am immensely grateful.

Bibliography

[Abr70] Philip Samuel Abrams. *An APL Machine*. PhD thesis, Stanford University, Stanford, CA, USA, 1970.

[AFG+05] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[AH00] John Aycock and Nigel Horspool. Simple generation of static single-assignment form. pages 110–125. LNCS 1781, Springer Verlag, 2000.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[BBF+10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A trace-based JIT compiler for CIL. pages 708–725, 2010.

[BCE] BCEL homepage. http://jakarta.apache.org/bcel/.

[BCGF09] Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz. Stream-based dynamic compilation for object-oriented languages. In *TOOLS (47)*, pages 77–95, 2009.

[BCW+10] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. pages 59–68, 2010.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[BGF07] Michael Bebenita, Andreas Gal, and Michael Franz. Implementing fast jvm interpreters using java itself. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 145–154, New York, NY, USA, 2007. ACM.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. pages 169–190. ACM Press, 2006.

[BL96] Thomas Ball and James Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Paris, France, December 1996.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50:375–425, 2003.

[Bru10] Stefan Brunthaler. Inline caching meets quickening. pages 429–451, 2010.

[CBY+07] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference, 2007.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM.

[DP73] RJ Dakin and P.C. Poole. A mixed code approach. *The Computer Journal*, 16(3):219, 1973.

[DR05] Dibyendu Das and U. Ramakrishna. A practical and fast iterative algorithm for phi-function computation using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 27:426–440, 2005.

[EG03] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, November 2003.

[Ell84] J.R. Ellis. *A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1984.

[Fre98] Stephen N. Freund. The costs and benefits of java bytecode subroutines. In *In Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[Gal06] Andreas Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine, 2006.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Michael Bebenita, Mason Chang, Michael Franz, Edwin Smith, Rick Reitmaier, and Mohammad Haghighat. Trace-based just-in-time type specialization for dynamic languages. pages 465–478, 2009.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association.

[GP11] Shu-yu Guo and Jens Palsberg. The essence of compiling with traces. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 563–574, New York, NY, USA, 2011. ACM.

[GPF06] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Han74] Gilbert Joseph Hansen. *Adaptive systems for the dynamic run-time optimization of programs.* PhD thesis, Pittsburgh, PA, USA, 1974. AAI7420364.

[HCU91] Urs Hölzle, Craig Chambers, and David M. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991 (ECOOP '91)*, volume 512/1991 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1991.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[Höl94] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Stanford, CA, USA, 1994.

[HPG03a] J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[HPG03b] J.L. Hennessy, D.A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*, pages 253–279. Morgan Kaufmann, 2003.

[HWI+11] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. 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 '11, pages 405–418, New York, NY, USA, 2011. ACM.

[IBM] IBM. The system loader. *University of Michigan. Executive System for the IBM 7090 Computer, Vol. 1.*

[IHWN11] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 246–256. IEEE, 2011.

[Jyt] Jython. Jython - Python for Java. http://www.jython.org.

[KCSL00] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.

[Ler03] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[Mat08] Bernd Mathiske. The maxine vm, 9 2008.

[McC62] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[MCH99] JA Mathew, P.D. Coddington, and KA Hawick. Analysis and Development of Java Grande Benchmarks. *Proceedings of the ACM Java Grande Conference, San Francisco, CA, June*, 1999.

[MG01] Erik Meijer and John Gough. Technical Overview of the Common Language Runtime. http://research.microsoft.com/~emeijer/papers/clr.pdf, 2001.

[MH08] Duane Merrill and Kim Hazelwood. Trace fragment selection within method-based JVMs. pages 41–50, 2008.

[Mic02] Microsoft Corporation. The Microsoft .NET platform, 2002. http://www.microsoft.com/net/.

[Mit70] James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Pittsburgh, PA, USA, 1970. AAI7104538.

[MMM10] Andreas Jaeger Michael Matz, Jan Hubicka and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. AMD, 2010. http://x86-64.org/documentation/abi.pdf.

[Moza] Mozilla. Rhino - JavaScript for Java. http://www.mozilla.org/rhino.

[Mozb] Mozilla. Tamarin Project - http://www.mozilla.org/projects/tamarin/ - August 1, 2007.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing Direct Threaded Code by Selective Inlining. pages 291–300, New York, NY, USA, 1998. ACM.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[SGC01] Dave Sager, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[Sta08] Standard Performance Evaluation Corporation. *SPECjvm2008*, 2008. http://www.spec.org/jvm2008/.

[Str93] Bjarne Stroustrup. *The Evolution of C++: 1985-1989*. MIT Press, Cambridge, MA, USA, 1993.

[SYN06] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, 2006.

[TIO11] TIOBE. TIOBE Index of Language Popularity. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 9 2011.

[TWSC10] Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. Improving compiler-runtime separation with XIR. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 39–50. ACM Press, 2010.

[WCB+09] Christian Wimmer, Marcelo S. Cintra, Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz. Phase detection using trace compilation. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 172–181, New York, NY, USA, 2009. ACM.

[WGF08] Gregor Wagner, Andreas Gal, and Michael Franz. Slim vm: optimistic partial program loading for connected embedded java virtual machines. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, PPPJ '08, pages 117–126, New York, NY, USA, 2008. ACM.

[Wha01] John Whaley. Partial method compilation using dynamic profile information. pages 166–179, 2001.

[WM05] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 132–141, New York, NY, USA, 2005. ACM.

[YW06] Hiroshi Yamauchi and Mario Wolczko. Writing Solaris device drivers in Java. In *PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, page 3, New York, NY, USA, 2006. ACM Press.

[ZBS07] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. Yeti: a gradually extensible trace interpreter. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 83–93. ACM Press, 2007.