YETI: A GRADUALLY EXTENSIBLE TRACE INTERPRETER

by

Mathew Zaleski

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2007 by Mathew Zaleski

Abstract

YETI: a graduallY Extensible Trace Interpreter

Mathew Zaleski

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2007

The design of new programming languages benefits from interpretation, which can provide a simple initial implementation, flexibility to explore new language features, and portability to many platforms. The only downside is speed of execution, as there remains a large performance gap between even efficient interpreters and mixed-mode systems that include a just-in-time (JIT) compiler. Augmenting an interpreter with a JIT, however, is not a small task. Today, Java JITs are loosely-coupled with the interpreter, with callsites of methods being the only transition point between interpreted and native code. To compile whole methods, the JIT must duplicate a sizable amount of functionality already provided by the interpreter, leading to a "big bang" development effort before the JIT can be deployed. Instead, adding a JIT to an interpreter would be easier if it were possible to leverage the existing functionality.

First, we show that packaging virtual instructions as lightweight callable routines is an efficient way to build an interpreter. Then, we describe how callable bodies help our interpreter to efficiently identify and run traces. Our closely coupled dynamic compiler can fall back on the interpreter in various ways, permitting a incremental approach in which additional performance gains can be realized as it is extended in two dimensions: (i) generating code for more types of virtual instructions, and (ii) identifying larger compilation units. Currently, Yeti identifies straight line regions of code and traces, and generates non-optimized code for roughly 50 Java integer and object bytecodes. Yeti runs roughly twice as fast as a direct-threaded interpreter on SPECjvm98 benchmarks.

Acknowledgements

thanks all yous guys.

Contents

1 Introduction **1**

1.1 Challenges of Evolving to a Mixed-Mode System . . . 3

1.2 Challenges of Efficient Interpretation . . . 4

1.3 What We Need . . . 4

1.4 Overview of Our Solution . . . 5

1.5 Thesis Statement . . . 7

1.6 Contributions . . . 8

1.7 Outline of Thesis . . . 9

2 Background **11**

2.1 High Level Language Virtual Machine . . . 11

2.1.1 Overview of a Virtual Program . . . 13

2.1.2 Interpretation . . . 14

2.1.3 Early Just in Time Compilers . . . 15

2.2 Challenges to HLL VM Performance . . . 16

2.2.1 Polymorphism and the Implications of Object-oriented Programming . 17

2.2.2 Late binding . . . 20

2.3 Early Dynamic Optimization . . . 21

2.3.1 Manual Dynamic Optimization . . . 21

2.3.2 Application specific dynamic compilation . . . 21

2.3.3 Dynamic Compilation of Manually Identified Static Regions 22

2.4 Dynamic Object-oriented optimization 23

2.4.1 Finding the destination of a polymorphic callsite 23

2.4.2 Smalltalk and Self 25

2.4.3 Java JIT as Dynamic Optimizer 27

2.4.4 JIT Compiling Partial Methods 28

2.5 Traces 29

2.6 Hotpath 31

2.7 Chapter Summary 32

3 Dispatch Techniques **33**

3.1 Switch Dispatch 34

3.2 Direct Call Threading 36

3.3 Direct Threading 36

3.4 Dynamic Hardware Branch Prediction 38

3.5 The Context Problem 39

3.6 Subroutine Threading 40

3.7 Optimizing Dispatch 42

3.7.1 Superinstructions 42

3.7.2 Selective Inlining 42

3.7.3 Replication 44

3.8 Chapter Summary 44

4 Design and Implementation of Efficient Interpretation **45**

4.1 Understanding Branches 47

4.2 Handling Linear Dispatch 48

4.3 Handling Virtual Branches 50

4.4 Handling Virtual Call and Return 53

4.5 Chapter Summary . . . 56

5 Evaluation of Context Threading **59**

5.1 Experimental Set-up . . . 60

5.1.1 Virtual Machines and Benchmarks . . . 60

5.1.2 Performance and Pipeline Hazard Measurements . . . 62

5.2 Interpreting the data . . . 63

5.2.1 Effect on Pipeline Branch Hazards . . . 67

5.2.2 Performance . . . 68

5.3 Inlining . . . 72

5.4 Limitations of Context Threading . . . 74

5.5 Chapter Summary . . . 76

6 Design and Implementation of YETI **79**

6.1 Structure and Overview of Yeti . . . 80

6.2 Region Selection . . . 84

6.2.1 Initiating Region Discovery . . . 84

6.2.2 Linear Block Detection . . . 85

6.2.3 Trace Selection . . . 87

6.3 Trace Exit Runtime . . . 89

6.3.1 Trace Linking . . . 90

6.4 Generating code for traces . . . 91

6.4.1 Interpreted Traces . . . 92

6.4.2 JIT Compiled Traces . . . 93

6.4.3 Trace Optimization . . . 97

6.5 Other implementation details . . . 99

6.6 Chapter Summary . . . 100

7 **Evaluation of Yeti** **103**

7.1 Experimental Set-up 104

7.2 Effect of region shape on dispatch 107

7.3 Effect of region shape on performance 111

7.4 Early Pentium Results 117

7.5 Identification of Stall Cycles 119

7.5.1 GPUL 119

7.5.2 GPUL results 120

7.5.3 Trends 122

7.6 Chapter Summary 126

8 **Conclusions and Future Work** **129**

8.1 Conclusions and Lessons Learned 129

8.2 Future work 132

8.2.1 Virtual instruction bodies as nested functions 132

8.2.2 Extension to Dynamically Typed Languages 133

8.2.3 New shapes of region body 134

8.2.4 Vision for new language implementation 135

8.3 Elevator pitch 135

Bibliography **137**

Chapter 1

Introduction

Modern computer languages are commonly implemented in two main parts – a compiler that targets a virtual instruction set, and a so-called *high-level language virtual machine* (or simply language VM) to run the resulting virtual program. This approach simplifies the compiler by eliminating the need for any machine dependent code generation. Tailoring the virtual instruction set can further simplify the compiler by providing operations that perfectly match the functionality of the language.

There are two ways a language VM can run a virtual program. The simplest approach is to interpret the virtual program. An interpreter dispatches a *virtual instruction body* to emulate each virtual instruction in turn. A more complicated, but faster, approach deploys a dynamic, or just in time (JIT), compiler to translate the virtual instructions to machine instructions and dispatch the resulting native code. *Mixed-mode* systems interpret some parts of a virtual program and compile others. In general, compiled code will run much more quickly than virtual instructions can be interpreted. By judiciously choosing which parts of a virtual program to JIT compile, a mixed-mode system can run much more quickly than the fastest interpreter.

Currently, although many popular languages depend on virtual machines, relatively few JIT compilers have been deployed. Notable exceptions include research languages like Self and several Java Virtual Machines (JVM). Consequently, users of important computer languages,

including JavaScript, Python, and many others, do not enjoy the performance benefits of mixed-mode execution.

The primary goal of our research is to make it easier to extend an interpreter with a JIT compiler. To this end we describe a new architecture for a language VM that significantly increases the performance of interpretation at the same time as it reduces the complexity of deploying a mixed-mode system. Our technique has two main features.

First, our JIT identifies and compiles hot interprocedural paths, or traces. Traces are single entry multiple exit regions that are easier to compile than the methods compiled by current systems. In addition, hot traces help predict the destination of virtual branches. This means that even before traces are compiled they provide a simple way to improve the interpreted performance of virtual branches.

Second, we implement virtual instruction bodies as lightweight, callable routines at the same time as we closely integrate the JIT compiler and interpreter. This gives JIT developers a simple alternative to compiling each virtual instruction. Either a virtual instruction is translated to native code, or instead, a call to the corresponding body is generated. The task of JIT developers is thereby simplified by making it possible to deploy a fully functional JIT compiler that compiles only a subset of virtual instructions. In addition, callable virtual instruction bodies have a beneficial effect on interpreter performance because they enable a simple interpretation technique, subroutine threading, that very efficiently executes straight-line, or non-branching, regions of a virtual program.

We prototype our ideas in Java because there exist many high-quality Java interpreters and JIT compilers with which to compare our results. We are able to determine that the performance of our prototype compares favourably with state-of-the art interpreters like JamVM and SableVM. An obvious next step would be to apply our techniques to enhance the performance of languages that currently do not offer a JIT.

The discussion in the next few sections refers to many technical terms and techniques that are described in detail in Chapter 2, which introduces the basic concepts and related work, and

Chapter 3, which provides a tutorial-like description of several interpreter techniques.

1.1 Challenges of Evolving to a Mixed-Mode System

Today, the usual approach taken by mixed-mode systems is to identify frequently executed, or *hot*, methods. Hot methods are passed to the JIT compiler which compiles them to native code. Then, when the interpreter sees an invocation of a compiled method, it dispatches the native code instead.

Up Front Effort This method-oriented approach has been followed for many years, but requires a large up-front investment in effort. Such a system cannot improve the performance of a method until it can compile every feature of the language that appears in it. For significant applications this requires the JIT to compile the whole language, including complicated features already implemented by high-level virtual instruction bodies, such as those for method invocation, object creation, and exception handling.

Compiling Cold Code Just because a method is frequently executed does not mean that all the instructions within it are frequently executed also. In fact, regions of a hot method may be *cold*, that is, they may have never executed. Compiling cold code has more implications than simply wasting compile time. Except at the very highest levels of optimization, where analyzing cold code may prove useful facts about hot regions, there is little point compiling code that never runs. A more serious issue is that cold code increases the complexity of dynamic compilation. We give three examples. First, for late binding languages such as Java, cold code likely contains references to program values which are not yet bound. Thus, when the cold code does eventually run, the generated code and the runtime that supports it must deal with the complexities of late binding [72]. Second, certain dynamic optimizations are not possible without runtime profiling information. Foremost amongst these is the optimization of virtual function calls. Since there is no profiling information for cold code the JIT may have

to generate relatively slow, conservative code. This issue is even more important for languages like Python. Without runtime information a Python JIT may not know whether the inputs of simple arithmetic operations such as addition are integers, floats, or strings. Third, as execution proceeds, some of the formerly cold regions in compiled methods may become hot. The conservative assumptions made during the initial compilation may now be a drag on performance. The straightforward-sounding approach of recompiling the method containing the cold code is complicated by problems such as what to do about threads that are still executing in the method or that will return to the method in the future.

1.2 Challenges of Efficient Interpretation

After a virtual program is *loaded* by an interpreter into memory it can be executed by *dispatching* each virtual instruction body (or just *body*) in the order specified by the virtual program. This is not a typical workload because the control transfer from one body to the next is data dependent on the sequence of instructions making up the virtual program. This makes the dispatch branches hard for a processor to predict. Ertl and Gregg observed that the performance of otherwise efficient interpretation is limited by pipeline stalls and flushes due to extremely poor branch prediction [27].

1.3 What We Need

These considerations suggest that the architecture of a *gradually* extensible mixed-mode virtual machine should have three important properties.

1. Virtual bodies should be callable. This allows JIT implementors to compile only some instructions, and fall back on the emulation functionality already implemented by the virtual instruction bodies for others.

2. The unit of compilation must be dynamically determined and of flexible shape. This allows the JIT compiler to translate hot regions while avoiding cold code.

3. As new regions of hot code reveal themselves and are compiled, a way is needed of gracefully linking them on to previously compiled hot code.

Callable Virtual Instruction Bodies Packaging bodies as callable can also address the prediction problems observed in interpreters. When a virtual program is loaded, every straight-line sequence of virtual instructions can be translated to a very simple sequence of generated machine instructions. Corresponding to each virtual instruction we generate a single direct call machine instruction which dispatches the corresponding virtual instruction body. Executing the resulting generated code thus emulates each virtual instruction in the linear sequence in turn. No branch mispredictions occur because the destination of each direct call is explicit and the return instruction ending each body is predicted perfectly by the return branch predictor present in most modern processors.

Traces Our system compiles frequently executed, dynamically identified interprocedural paths, or traces. Traces contain no cold code, so our system leaves all the complexities of running cold code to the interpreter. Since traces are paths through the virtual program, they explicitly predict the destination of each virtual branch. As a consequence even a very simple implementation of traces can significantly improve performance by reducing branch mispredictions caused by dispatching virtual branches.

1.4 Overview of Our Solution

In this dissertation we describe a system that supports dynamic compilation units of varying shapes. Just as a virtual instruction body implements a virtual instruction, a *region body* implements a region of the virtual program. Possible region bodies include single virtual instruc-

tions, basic blocks, methods, partial methods, inlined method nests, and traces (i.e., frequently-executed paths through the virtual program). The key idea is to package every region body as callable, regardless of the size or shape of the region of the virtual program that it implements. The interpreter can then execute the virtual program by dispatching each region body in sequence.

Region bodies corresponding to longer sequences of virtual instructions will run faster than those compiled from short ones because fewer dispatches are required. In addition, larger region bodies should offer more opportunities for optimization. However, larger region bodies are more complicated and so we expect them to require more development effort to detect and compile than short ones. This suggests that the performance of a mixed-mode VM can be gradually extended by incrementally increasing the scope of region bodies it identifies and compiles. Ultimately, the peak performance of the system should be at least as high as current method-based JIT compilers since, with basically the same engineering effort, inlined method nests could be compiled to region bodies also.

The practicality of our scheme depends on the efficiency of dispatching bodies by calling them. Thus, the first phase of our research, described in Chapters 4 and 5, was to retrofit SableVM [31], a Java virtual machine, and `ocamlrun`, an OCaml interpreter [13], to a new hybrid dispatch technique we call *context threading*. We evaluated context threading on PowerPC and Pentium 4 platforms by comparing branch predictor and runtime performance of common benchmarks to unmodified, direct-threaded, versions of the virtual machines. We show that callable bodies can be dispatched more efficiently than dispatch techniques currently thought to be very efficient. However, it proved difficult to cleanly add trace detection and profiling instrumentation to our implementation of context threading. Consequently, to build our trace-based JIT we decided to start afresh.

In the second phase of this research, described in Chapters 6 and 7, we gradually extended JamVM, a cleanly implemented and relatively high performance Java interpreter [52], with a trace oriented JIT compiler. We built Yeti, (graduallY Extensible Trace Interpreter) in five

stages with the explicit intention of providing a design trajectory from a simple system to a high performance implementation. First, we repackaged all virtual instruction bodies as callable. Our initial implementation executed only single virtual instructions which were dispatched via an indirect call from a simple dispatch loop. This is slow compared to context threading but very easy to instrument. Second, we identified *linear blocks*, or sequences of virtual instructions ending in branches. Third, we extended our system to identify and dispatch interpreted *traces*, or sequences of linear blocks. Traces are significantly more complex region bodies than linear blocks because they must accommodate virtual branch instructions. Fourth, we extended our trace runtime system to link traces together. In the fifth and final stage, we implemented a naive, non-optimizing compiler to compile the traces. An interesting feature of the JIT is that it performs simple compilation and register allocation for some virtual instructions but falls back on calling virtual instruction bodies for others. Our compiler currently generates PowerPC code for about 50 integer and object virtual instructions.

We chose traces because they have several attractive properties: (i) they can extend across the invocation and return of methods, and thus have an interprocedural view of the program, (ii) they contain only hot code, (iii) they are relatively simple to compile as they are *single-entry multiple-exit* regions of code, and (iv), it is straightforward to generate new traces and link them onto existing ones as new hot paths reveal themselves.

Instrumentation built into our prototype shows that, on the average, traces accurately predict paths taken by the Java SPECjvm98 benchmark programs. Performance measurements show that the overhead of trace identification is reasonable. Even with our naive compiler, Yeti runs about twice as fast as unmodified JamVM.

1.5 Thesis Statement

The implementation of a new high-level language virtual machine should be extensible to a high performance mixed-mode system as the language matures. To achieve this, an interpreter

should be designed to dispatch virtual instructions by calling them. This achieves efficient dispatch, and hence high performance interpretation, by making it easy to eliminate branch mispredictions caused by the dispatch of straight-line virtual code. Callable virtual instruction bodies also facilitate extending the interpreter with a JIT compiler because the bodies can be called from generated code. The unit of compilation translated by the JIT compiler should be a dynamically identified region containing only hot code. Hot interprocedural paths, or traces, are a good choice because they are simple to compile and link together. Since hot traces predict the destination of virtual branch instructions they can also be used to improve the interpretation performance of virtual branch instructions. Thus, a trace-based interpreter performs better than current interpreter techniques and also is more easily extended with a JIT compiler.

1.6 Contributions

The contributions of this thesis are twofold:

1. We show that organizing an interpreter to call virtual instruction bodies is desirable on modern processors because the additional cost of call and return is more than made up for by improvements in branch prediction. We show that subroutine threading significantly outperforms direct threading, for Java and OCaml on Pentium and PowerPC. We show how with a few extensions a subroutine-threaded interpreter can perform as well as or better than the best reported interpretation techniques.

2. We propose an architecture for, and describe our implementation of, a trace-oriented JIT compiler. We show how to extend our interpreter to identify interprocedural paths, or traces through the program. We describe a novel design for a simple JIT compiler that compiles only a subset of the virtual instructions in each trace.

1.7 Outline of Thesis

We describe an architecture for a virtual machine interpreter that facilitates the gradual extension to a trace-based mixed-mode JIT compiler. We demonstrate the feasibility of this approach in a prototype, Yeti, and show that performance can be gradually improved as larger program regions are identified and compiled.

In Chapters 2 and 3 we present background and related work on interpreters and JIT compilers. In Chapter 4 we describe the design and implementation of context threading. Chapter 5 describes how we evaluated context threading. The design and implementation of Yeti is described in Chapter 6. We evaluate the benefits of this approach in Chapter 7. Finally, we discuss possible avenues for future work and conclude in Chapter 8.

Chapter 2

Background

Researchers have investigated how virtual machines should execute high-level language programs for many years. The research has been focused on a few main areas. First, innovative virtual machine support can play a role in the deployment of qualitatively new and different computer languages. Second, virtual machines provide an infrastructure by which ordinary computer languages can be more easily deployed on many different hardware platforms. Third, various techniques have been proposed that enable programs to run faster than before.

This chapter will describe research which touches on all these issues. We will briefly discuss interpretation in preparation for a more in-depth treatment in Chapter 3. We will describe how modern object-oriented languages depend on the virtual machine to efficiently invoke methods by following the evolution of this support from the early efforts to modern speculative inlining techniques. Finally, we will briefly describe trace-based binary optimization to set the scene for Chapter 6.

2.1 High Level Language Virtual Machine

A static compiler is probably the best solution when performance is paramount, portability is not a great concern, destinations of calls are known at compile time and programs bind to external symbols before running. Thus, most third generation languages like C and FORTRAN

are implemented this way. However, if the language is object-oriented, binds to external references late and must run on several platforms, it may be advantageous to implement a compiler that targets a fictitious *high-level language virtual machine* (HLL VM) instead.

In Smith's taxonomy, an HLL VM is a system that provides a process with an execution environment that does not correspond to any particular hardware platform [64]. The interface offered to the high-level language application process is usually designed to hide differences between the platforms to which the VM will eventually be ported. For instance, UCSD Pascal p-code [78, 16] and Java bytecode [51] both express virtual instructions as stack operations that take no register arguments. Gosling, one of the designers of the Java virtual machine, has said that he based the design of the JVM on the p-code machine [3]. Smalltalk [35], Self [73] and many other systems have taken a similar approach. This makes it easier to port the VM between hardware platforms that have variously sized register files. A VM may also provide virtual instructions that support peculiar or challenging features of the language. For instance, a Java virtual machine has specialized virtual instructions (`invokevirtual`, etc) in support of virtual method invocation. This allows the compiler to generate a single, relatively high-level virtual instruction instead of a sequence of complex machine and ABI dependent instructions.

This approach has benefits for the users as well. For instance, applications can be distributed in a platform neutral format. In the case of the Java class libraries or UCSD Pascal programs the amount of virtual software far exceeds the size of the VM. The advantage is that the relatively small amount of effort required to port the VM to a new platform enables a large body of virtual applications to run on the new platform also.

There are various approaches a HLL VM can take to actually execute a virtual program. An interpreter fetches, decodes, then emulates each virtual instruction in turn. Hence, interpreters are slow but can be very portable. Faster, but less portable, a dynamic compiler can translate to native code and dispatch regions of the virtual application. A dynamic compiler can exploit runtime knowledge of program values so it can sometimes do a better job of optimizing the program than a static compiler [67].

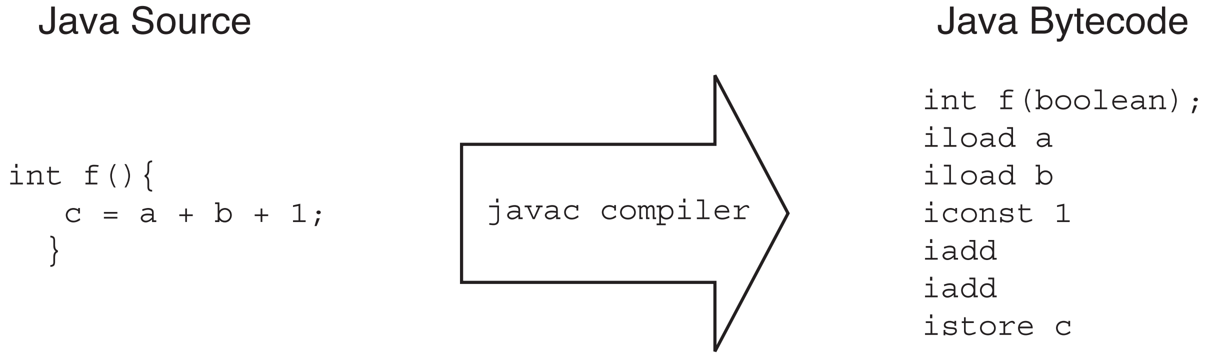


Figure 2.1: Example Java Virtual Program showing source (on the left) and Java virtual instructions, or bytecodes, on the right.

2.1.1 Overview of a Virtual Program

todo: make beefier example

A virtual program, as shown in Figure 2.1, is a sequence of virtual instructions and related meta-data. The figure introduces an example program we will use as a running example, so we will briefly describe it here. First, a compiler, `javac` in the example, creates a *class file* describing the virtual program in a standardized format. (We show only one method, but any real Java example would define a whole class.) Our example consists of just one Java expression `{c=a+b+1}` which adds the values of two Java local variables and a constant and stores the result in a third. The compiler has translated this to the sequence of virtual instructions shown on the right. The actual semantics of the virtual instructions are not important to our example other than to note that none are virtual branch instructions.

The distinction between a virtual instruction and an *instance* of a virtual instruction is conceptually simple but sometimes hard to clearly distinguish in prose. We will always refer to a specific use of a virtual instruction as an "instance". For example, the first instruction in our example program is an instance of `iload`. On the other hand, we might also use the term virtual instruction to refer to a kind of operation, for example that the `iload` virtual instruction takes one parameter.

Java virtual instructions may take implicit arguments that are passed on a expression stack. For instance, in Figure 2.1, the `iadd` instruction pops the top two slots of the expression stack and pushes their sum. This style of instruction set is very compact because there is no need to

explicitly list parameters of most virtual instructions. Consequently many virtual instructions, like `iadd`, consist of only the opcode. Since there are fewer than 256 Java virtual instructions, the opcode fits in a byte, and so Java virtual instructions are often referred to as *bytecode*.

In addition to arguments passed implicitly on the stack, certain virtual instructions take immediate operands. In our example, the `iconst` virtual instruction takes an immediate operand of 1. Immediate operands are also required by virtual branch instructions (the offset of the destination) and by various instructions used to access data.

The bytecode in the figure depends on a stack frame organization that distinguishes between local variables and the expression stack. *Local variable array* slots, or *lva* slots, are used to store local variables and parameters. The simple function shown in the figure needs only four local variable slots. The first slot, lva[0], stores a hidden parameter, the object handle[1] to the invoked-upon object and is not used in this example. Subsequent slots, lva[1], lva[2] and lva[3] store `a`, `b` and `c` respectively. The expression stack is used to store temporaries for most calculations and parameter passing. In general "load" form bytecodes push values in lva slots onto the expression stack. Bytecodes with "store" in their mnemonic typically pop the value on top of the expression stack and store it in a named lva slot.

2.1.2 Interpretation

An interpreter is the simplest way for an HLL VM to execute a guest virtual program. Whereas the persistent format of a virtual program conforms to some external specification, when it is read by an interpreter the structure of its *loaded representation* is chosen by the designers of the interpreter. For instance, designers may prefer a representation that word-aligns all immediate parameters regardless of their size. This would be less compact, but more portable and potentially faster to access, than the original byte code on most architectures.

An abstraction implemented by most interpreters is the notion of a *virtual program counter*, or `vPC`. It points into the loaded representation of the program and serves two main purposes.

[1] lva[0] stores the local variable known as `this` to Java (and C++) programmers.

First, the vPC is used by dispatch code to indicate where in the virtual program execution has reached and hence which virtual instruction to emulate next. Second, the vPC is conventionally referred to by virtual instruction bodies to access immediate operands.

Interpretation is not efficient

We do not expect interpretation to be efficient compared to executing compiled native code. Consider Java's iadd virtual instruction. On a typical processor an integer add can be performed in one instruction. To emulate a virtual addition instruction requires three or more additional instructions to load the inputs from and store the result to the expression stack.

However, it is not just the path length of emulation that causes performance problems. Also important is the latency of the branch instructions used to transfer control to the virtual instruction body. To optimize dispatch researchers have proposed various *dispatch* techniques to efficiently branch from body to body. Recently, Ertl and Gregg showed that on modern processors branch mispredictions caused by dispatch branches are a serious drain on performance [27, 28].

When emulated by most current high-level language virtual machines, the branching patterns of the virtual program are hidden from the branch prediction resources of the underlying real processor. This is despite the fact that a typical virtual machine defines roughly the same sorts of branch instructions as does a real processor — and that a running virtual program exhibits similar patterns of virtual branch behaviour as does a native program running on a real CPU. In Section 3.5 we discuss in detail how our approach to dispatch deals with this issue, which we have dubbed the *context problem.*

2.1.3 Early Just in Time Compilers

A faster way of executing a guest virtual program is to compile its virtual instructions to native code before it is executed. This approach long predates Java, perhaps first appearing for APL on the HP3000 [47] as early as 1979. Deutsch and Schiffman [23] built an early just in time

(JIT) compiler for Smalltalk that ran about twice as fast as interpretation.

Early systems were highly memory constrained by modern standards. It was of great concern, therefore, when translated native code was found to be about four times larger than the originating bytecode[2]. Lacking virtual memory, Deutsch and Schiffman took the view that dynamic translation of bytecode was a space time trade-off. If space was tight then native code (space) could be released at the expense of re-translation (time). Nevertheless, their approach was to execute only native code. Each method had to be fetched from a native code cache or else re-translated before execution. Today a similar attitude prevails except that it has also been recognized that some code is so infrequently executed that it need not be translated in the first place. The bytecode of methods that are not hot can simply be interpreted.

A JIT can improve the performance of a JVM substantially. Relatively early Java JIT compilers from Sum Microsystems, as reported by the development team in 1997, improved the performance of the Java `raytrace` application by a factor of 2.2 and `compress` by 6.8 [18][3]. More recent JIT compilers have increased the performance further [2, 4, 69]. For instance, on a modern personal computer Sun's Hotspot server dynamic compiler currently runs the entire SPECjvm98 suite more than 4 times faster than the fastest interpreter. Some experts suggest that in the not too distant future, systems based on dynamic compilers will run *faster* than the code generated by static compilers [67, 66, slide 28].

2.2 Challenges to HLL VM Performance

Modern languages offer users powerful features that challenge VM implementors. In this section we will discuss the impact of object-oriented method invocation and late binding of external references. There are many other issues that affect Java performance which we discuss

[2]This is less than one might fear given that on a RISC machine one typical arithmetic bytecode will be naïvely translated into two loads (pops) from the expression stack, one register-to-register arithmetic instruction to do the real work and a store (push) back to the new top of the expression stack.

[3]These benchmarks are singled out because they eventually were adopted by the SPEC consortium to be part of the SPECjvm98 [65] benchmark suite.

only briefly. The most important amongst them are memory management and thread synchronization.

Garbage collection refers to a set of techniques used to manage memory in Java (as in Smalltalk and Self). In general the idea is that unused memory (garbage) is detected automatically by the system. As a result the programmer is relieved of any responsibility for freeing memory that he or she has allocated. Garbage collection techniques are somewhat independent of dynamic compilation techniques. The primary interaction requires that threads can be stopped in a well-defined state prior to garbage collection. So-called *safe points* must be defined at which a thread periodically saves its state to memory. Code generated by a JIT compiler must ensure that safe points occur frequently enough that garbage collection is not unduly delayed. Typically this means that each transit of a loop must contain at least one safe point.

Java provides explicit, built-in, support for threads. *Thread synchronization* refers mostly to the functionality that allows only one thread to enter certain regions of code at a time. Thread synchronization must be implemented at various points and the techniques for implementing it must be supported by code generated by the JIT compiler.

2.2.1 Polymorphism and the Implications of Object-oriented Programming

Over the last few decades object-oriented development grew from vision, to an industry trend, to a standard programming tool. Object-oriented techniques stressed development systems in many ways, but the one we need to examine in detail here is the challenge of polymorphic method invocation.

The destination of a callsite in an object-oriented language is not determined solely by the signature of a method, as in C or FORTRAN. Instead, it is determined at run time by a combination of the method signature and the class of the invoked-upon object. Callsites are said to be *polymorphic* as the invoked-upon object may turn out to be one of potentially many classes.

```
void sample(Object[] otab){
   for(int i=0; i<otab.length; i++){
      otab[i].toString();  //polymorphic callsite
      }
   }
```

Figure 2.2: Example of Java method containing a polymorphic callsite

Most object-oriented languages categorize objects into a hierarchy of *classes*. Each object is an *instance* of a class which means that the methods and data fields defined by that class are available for the object. Each class, except the root class, has a *super-class* or *base-class* from which it *inherits* fields and methods.

Each class may override a method and so at run time the system must dispatch the definition of the method corresponding to the class of the invoked-upon object. In many cases it is not possible to deduce the exact type of the object at compile time.

A simple example will make the above description concrete. When it is time to debug a program almost all programmers rely on facilities to view a textual description of their data. In an object-oriented environment this suggests that each object should define a method that returns a string description of itself. This need was recognized by the designers of Java and consequently they defined a method in the root class `Object`:

```
public String toString()
```

to serve this purpose. The `toString`[4] method can be invoked on every Java object. Consider an array of objects in Java. Suppose we code a loop that iterates over the array and invokes the `toString` method on each element as in Figure 2.2.

There are literally hundreds of definitions of `toString` in a Java system and in many cases the compiler cannot discern which one will be the destination of the callsite. Since it is not possible to determine the destination of the callsite at compile time it must be done when the program executes. Determining the destination taxes performance in two main ways.

[4] It is the text returned by `toString` that appears in various views of an interactive debugger

First, locating the method to dispatch at run time requires computation. This will be discussed in Section 2.4.1. Second, the inability to predict the destination of a callsite at compile time reduces the efficacy of interprocedural optimizations and thus results in relatively slow systems. This is discussed next.

Impact of Polymorphism on Optimization

Optimization can be stymied by polymorphic callsites. At compile time, an optimizer cannot determine the destination of a call, so obviously the target cannot be inlined. In fact, standard interprocedural optimization as carried out by an optimizing C or FORTRAN compiler is simply not possible [54].

In the absence of interprocedural information, an optimizer cannot guess what calculations are made by a polymorphic callee. Knowledge of the destination of the callsite would permit a more precise analysis of the values modified by the call. For instance, with runtime information, the optimizer may know that only one specific version of the method exists and that this definition simply returns a constant value. Code compiled speculatively under the assumption that the callsite remains monomorphic could constant propagate the return value forward and hence be much better than code compiled under the conservative assumption that other definitions of the method may be called.

Given the tendency of modern object-oriented software to be factored into many small methods which are called throughout a program, even in its innermost loops, these optimization barriers can significantly degrade the quality of code produced. A typical example might be that common subexpression elimination cannot combine identical memory accesses separated by a polymorphic callsite because it cannot prove that all possible callees do not kill the memory location. To achieve performance comparable to procedural compiled languages, interprocedural optimization techniques must somehow be applied to regions laced with polymorphic callsites.

Section 2.4 describes various solutions to these issues.

2.2.2 Late binding

A basic design issue for any language is when external references are resolved. Java binds references very late in order to support flexible packaging in general and downloadable code in particular. (This contrasts with traditional languages like C, which rely on a link-editor to bind to external symbols before they run.) The general idea is that a Java program may start running before all the classes that it needs are locally available. In Java, binding is postponed until the last possible moment, when the virtual instruction making the reference executes for the first time. Then, during the first execution, the reference is either resolved or a software exception is raised. This means that the references a program attempts to resolve depends on the path of execution through the code.

This approach is convenient for users and challenging for language implementors. Whenever Java code is executed for the first time the system must be prepared to handle unresolved external references. An obvious, but slow, approach is to simply check whether an external reference is resolved each time the virtual instruction executes. For good performance, only the first execution should be burdened with any binding overhead. One way to achieve this is for the virtual program to rewrite itself when an external reference is resolved. For instance, suppose a virtual instruction, `vop`, takes an immediate parameter that names an unresolved class or method. When the virtual instruction is first executed the external name is resolved and an internal VM data structure describing it is created. The loaded representation of the virtual instruction is then rewritten, say to `vop_resolved`, which takes the address of the data structure as an immediate parameter. The implementation of `vop_resolved` can safely assume that the external reference has been resolved successfully. Subsequently `vop_resolved` will execute in place of `vop` with no binding overhead. [5]

The process of virtual instruction rewriting is relatively simple to carry out when instructions are being interpreted. For instance, it is possible to fall back on standard thread support

[5] This roughly describes how JamVM and SableVM handle late binding.

libraries to protect overwriting from multiple threads racing to rewrite the instruction. It is more challenging if the resolution is being carried out by dynamically compiled native code [72].

2.3 Early Dynamic Optimization

Early efforts to build dynamic optimizers were embedded in applications or C or FORTRAN run time systems.

2.3.1 Manual Dynamic Optimization

Early experiments with dynamic optimization indicated that large performance improvements are possible. Typical early systems were application-specific. Rather than compile a language, they dynamically generated machine code to calculate the solution to a problem described by application specific data. Later, researchers built semi-automatic dynamic systems that would re-optimize regions of C programs at run time [50, 5, 33, 37, 36].

Although the semi-automatic systems did not enable dramatic performance improvements across the board, this may be a consequence of the performance baseline to which they compared themselves. The prevalent programming languages of the time were supported by static compilation and so it was natural to use the performance of highly optimized binaries as the baseline. The situation for modern languages like Java is somewhat different. Dynamic techniques that do not pay off relative to statically optimized C code may be beneficial when applied to code naïvely generated by a JIT. Consequently, a short description of a few early systems seems worthwhile.

2.3.2 Application specific dynamic compilation

In 1968 Ken Thompson built a dynamic compiler which accepted a textual description of a regular expression and dynamically translated it into machine code for an IBM 7094 computer [48]. The resulting code was dispatched to find matches quickly.

In 1985 Pike et al. invented an often-cited technique to generate good code for quickly copying, or bitblt'ing, regions of pixels from memory onto a display [56]. They observed that there was a bewildering number of special cases (caused by various alignments of pixels in display memory) to consider when writing a good general purpose bitblit routine. Instead they wrote a dynamic code generator that could produce a good (near optimal) set of machine instructions for each special case. At worst their system executed only about 400 instructions to generate code for a bitblit.

2.3.3 Dynamic Compilation of Manually Identified Static Regions

In the mid-1990's Lee and Leone [50] built FABIUS, a dynamic optimization system for the research language ML [33]. FABIUS depends on a particular use of *curried functions.* Curried functions take one or more functions as parameters and return a new function that is a composition of the parameters. FABIUS interprets the call of a function returned by a curried function as a clue from the programmer that dynamic re-optimization should be carried out. Their results, which they describe as preliminary, indicate that small, special purpose, applications such as sparse matrix multiply or a network packet filter may benefit from their technique but the time and memory costs of re-optimization are difficult to recoup in general purpose code.

More recently it has been suggested that C and FORTRAN programs can benefit from dynamic optimization. Auslander et al [5], Grant et al [37, 36] and others have built semi-automatic systems to investigate this. Initially these systems required the user to identify regions of the program that should be dynamically re-optimized as well as the variables that are runtime constant. Later systems allowed the user to identify only the program variables that are runtime constants and could automatically identify which regions should be re-optimized at run time.

In either case, the general idea is that the user indicates regions of the program that may be beneficial to dynamically compile at run time. The dynamic region is precompiled into

template code. Then, at run time, the values of runtime constants can be substituted into the template and the dynamic region re-optimized. Auslander's system worked only on relatively small kernels like matrix multiply and quicksort. A good way to look at the results was in terms of *break even point*. In this view, the kernels reported by Auslander had to execute from about one thousand to a few tens of thousand of times before the improvement in execution time obtained by the dynamic optimization outweighed the time spent re-compiling and re-optimizing.

Subsequent work by Grant et al. created the DyC system [37, 36]. DyC simplified the process of identifying regions and applied more elaborate optimizations at run time. This system can handle real programs, although even the streamlined process of manually designating only runtime constants is reported to be time consuming. Their methodology allowed them to evaluate the impact of different optimizations independently, including complete loop unrolling, dynamic zero and copy propagation, dynamic reduction of strength and dynamic dead assignment elimination to name a few. Their results showed that only loop unrolling had sufficient impact to speed up real programs and in fact without loop unrolling there would have been no overall speedup at all.

2.4 Dynamic Object-oriented optimization

2.4.1 Finding the destination of a polymorphic callsite

Locating the definition of a method for a given object at run time is a search problem. To search for a method definition corresponding to a given object the system must search the classes in the hierarchy. The search starts at the class of the object, proceeds to its super class, to the super class of its super class, and so on, until the root of the class hierarchy is reached. If each method invocation requires the search to be repeated, the process will be a significant tax on overall performance. Nevertheless, this is exactly what occurs in a naïve implementation of Smalltalk, Self , Java, JavaScript or Python.

If the language permits early binding, the search may be converted to a table lookup at compile-time. For instance, in C++, all the possible destinations of a callsite are known when the program is loaded. As a result a C++ virtual callsite can be implemented as an indirect branch via a virtual table specific to the class of the object invoked on. This reduces the cost to little more than a function pointer call in C. The construction and performance of virtual function tables has been heavily studied, for instance by Driesen [24].

Real programs tend to have low *effective polymorphism*. This means that the average callsite has very few actual destinations. If fact, most callsites are *effectively monomorphic*, meaning they always call the same method. Note that low effective polymorphism does not imply that a smart compiler should have been able to deduce the destination of the call. Rather, it is a statistical observation that real programs typically make less use of polymorphism than they might.

Inlined Caching and Polymorphic Inlined Caching

For late-binding languages it is seldom possible to generate efficient code for a callsite at compile time. In response, various researchers have investigated how it might be done at run time. In general, it pays to cache the destination of a callsite when the callsite is commonly executed and its effective polymorphism is low. The *in-line cache*, invented by Deutsch and Schiffman [23] for Smalltalk more than 20 years ago, replaces the polymorphic callsite with the native instruction to call the cached method. The prologue of all methods is extended with fix-up code in case the cached destination is not correct. Deutsch and Shiffman reported hitting the in-line cache about 95% of the time for a set of Smalltalk programs.

Hölzle [42] extended the in-line cache to be a *polymorphic in-line cache* (PIC) by generating code that successively compares the class of the invoked object to a few possible destination types. The implementation is more difficult than an in-line cache because the dynamically generated native code sequence must sequentially compare and conditionally branch against several possible destinations. A PIC extends the performance benefits of an in-line cache to

effectively polymorphic callsites. For example, on a SPARCstation-2 Hölzle's lookup would cost only 8 + 2n cycles, where n is the actual polymorphism of the callsite. A PIC lookup costs little more than an in-line cache for effectively monomorphic callsites and is much faster for callsites that are effectively polymorphic.

2.4.2 Smalltalk and Self

Smalltalk adopted the position that essentially every software entity should be represented as an object. A fascinating discussion of the qualitative benefits anticipated from this approach appears in Goldberg's book [34].

The designers of Self took an even more extreme position. They held that even control flow should be expressed using object-oriented concepts. [6] They understood that this approach would require them to invent new ways to efficiently optimize message invocation if the performance of their system was to be reasonable. Their research program was extremely ambitious and they explicitly compared the performance of their system to optimized C code executing the same algorithms.

In addition, the Self system aimed to support the most interactive programming environment possible. Self supports debugging, editing and recompiling methods while a program is running with no need to restart. This requires very late binding. The combination of the radically pure object-oriented approach and the ambitious goals regarding development environment made Self a sort of trial-by-fire for object-oriented dynamic compilation techniques.

Ungar, Chambers and Hölzle have published several papers [14, 43, 42, 44] that describe how the performance of Self was increased from more than an order of magnitude slower than compiled C to only twice as slow. A readable summary of the techniques are given by Ungar et al [73]. A thumbnail summary would be that effective monomorphism can be exploited by a combination of type-checking guard code (to ensure that some object's type really is

[6] In Self, two blocks of code are passed as parameters to an if-else message sent to a boolean object. If the object is true the first block is evaluated, otherwise the second.

known) and static inlining (to expose the guarded code to interprocedural optimization). To give the flavor of this work we will briefly describe two specific optimizations: customization and splitting.

Customization

Customization is a relatively old object-oriented optimization introduced by Craig Chambers in his dissertation [14] in 1988. The general idea is that a polymorphic callsite can be turned into a static callsite (or inlined code) when the type of object on which the method is invoked is known. The approach taken by a customizing compiler is to replicate methods with type specialized copies so as to produce callsites where types are known.

Ungar et al. give a simple, convincing example in [73]. In Self, it is usual to write generic code, for instance algorithms that can be shared by integer and floating point code. An example is a method to calculate minimum. The `min` method is defined by a class called `Magnitude`. All concrete number classes, like `Integer` and `Float`, thus inherit the `min` method. A customizing compiler will arrange that customized definitions of `min` are compiled for `Integer` and `Float`. Inlining the customized methods replaces the polymorphic call[7] to < within the original `min` method by the appropriate arithmetic compare instructions[8] in each of the customized versions of integer and float `min`.

Method Splitting

Oftentimes, customized code can be inlined only when protected by a type guard. The guard code is essentially an if-then-else construct where the “if” tests the type of an object, the “then” inlines the customized code and the “else” performs the original polymorphic method invocation of the method. Chambers [14] noted that the predicate implemented by the guard establishes the type of the invoked object for one leg of the if-then-else, but following the merge

[7] In Self even integer comparison requires a message send.

[8] i.e. the integer customized version of `min` can issue an arithmetic integer compare and the float customization can issue a float comparison instruction.

point, this knowledge is lost. Hence, he suggested that following code be "split" into paths for which knowledge of types is retained. This suggests that instead of allowing control flow to merge after the guard, a splitting compiler can replicate following code to preserve type knowledge.

Incautious splitting could potentially cause exponential code size expansion. This implies that the technique is one that should only be applied to relatively small regions where it is known that polymorphic dispatch is hurting performance.

2.4.3 Java JIT as Dynamic Optimizer

The first Java JIT compilers translated methods into native instructions and improved polymorphic method dispatch by deploying techniques invented decades previously for Smalltalk. New innovations in garbage collection and thread synchronization, not discussed in this review, were also made. Despite all this effort, Java implementations were still slow. More aggressive optimizations had to be developed to accommodate the performance challenges posed by Java's object-oriented features, particularly the polymorphic dispatch of small methods. The writers of Sun's Hotspot compiler white paper note:

> In the Java language, most method invocations are *virtual* (potentially polymorphic), and are more frequently used than in C++. This means not only that method invocation performance is more dominant, but also that static compiler optimizations (especially global optimizations such as inlining) are much harder to perform for method invocations. Many traditional optimizations are most effective between calls, and the decreased distance between calls in the Java language can significantly reduce the effectiveness of such optimizations, since they have smaller sections of code to work with.[2, pp 17]

Observations similar to the above led Java researchers to perform speculative optimizations to transform the program in ways that are correct at some point, but may be invalidated by legal computations made by the program. For instance, Pechtchanski and Sarkar speculatively generate code for a method with only one loaded definition that assumes it will never be overridden. Later, if the loader loads a class that provides another definition of the method, the

speculative code may be incorrect and must not run again. In this case, the entire enclosing method (or inlined method nest) must be recompiled under more realistic assumptions and the original compilation discarded [55].

In principle, a similar approach can be taken if the speculative code is correct but turns out to be slower than it could be.

The infrastructure to replace a method is complex, but is a fundamental requirement of speculative optimization in a method-oriented dynamic compiler. It consists of roughly two parts. First, meta data must be produced when a method is optimized that allows local variables in the stack frame and registers of a running method to be migrated to a recompiled version. This is somewhat similar to the problem of debugging optimized code [43]. Later, at run time, the meta data is used to convert the stack frame of the invalid code to that of the recompiled code. Fink and Qian describe a technique called on stack replacement (OSR) [30] that shows how to restrict optimization so that recompilation is always possible. The key idea is that values that may be dead under traditional optimization schemes must be kept alive so that a less aggressively optimized replacement method can continue.

2.4.4 JIT Compiling Partial Methods

The dynamic compilers described thus far compile entire methods or inlined method nests. The problem with this approach is that even a hot method may contain cold code. The cold code may never be executed or perhaps will later become hot only after being compiled.

Compiling cold code that never executes can have only indirect effects such as allowing the optimizer to prove facts about the portions of the method that *are* hot. This can have a positive impact on performance, by enabling the optimizer to prove facts about hot regions that enable faster code to be produced. Also, it can have a negative impact, as the cold code may contain code that forces the optimizer to generate more conservative, slower, code for the hot regions.

Whaley described a prototype that compiled partial methods, skipping cold code. He modified the compiler to generate glue code stubs in the place of cold code. The glue code had

two purposes. First, to the optimizer at compile time, the glue code included annotations so that it appeared to use the same variables as the cold code. Consequently the optimizer has a true model of variables used in the cold regions and so generated correct code for the hot ones. Second, when run, the glue code interacted with the runtime system to exit the code cache and resume interpretation. Hence, if a cold region was entered control would simply revert to the interpreter. His results showed a large compile time savings, leading to modest speed ups for certain benchmarks [77].

Suganuma et al. investigated this issue further by modifying a method-based JIT to speculatively optimize hot inlined method nests. Their technique inlines only hot regions, replacing cold code with guard code [70]. The technique is speculative because conservative assumptions in the cold code are ignored. When execution triggers guard code it exposes the speculation as wrong and hence is a signal that continued execution of the inlined method nest may be incorrect. On stack replacement and recompilation are used to recover. They also measured a significant reduction in compile time. However, only a modest speedup was measured, suggesting either that conservative assumptions stemming from the cold code are not a serious concern or their recovery mechanism is too costly.

2.5 Traces

HP Dynamo [7, 25, 6] is a same-ISA binary optimizer. Dynamo initially interprets a binary executable program, detecting hot interprocedural paths, or *traces*, through the program as it runs. These traces are then optimized and loaded into a *trace cache*. Subsequently, when the interpreter encounters a program location for which a trace exists, it is dispatched from the trace cache. If execution diverges from the path taken when the trace was generated then a *trace exit* occurs, execution leaves the trace cache and interpretation resumes. If the program follows the same path repeatedly, it will be faster to execute code generated for the trace rather than the original code. Dynamo successfully reduced the execution time of many important

benchmarks. Several binary optimization systems, including DynamoRIO [12], Mojo [15], Transmeta's CMS [22], and others, have since used traces.

Dynamo uses a simple heuristic, called Next Executing Tail (NET), to identify traces. NET starts generating a trace from the destination of a hot reverse branch, since this location is likely to be the head of a loop, and hence a hot region of the program is likely to follow. If a given trace exit becomes hot, a new trace is generated starting from its destination.

Software trace caches are efficient structures for dynamic optimization. Bruening and Duesterwald [9] compare execution time coverage and code size for three dynamic optimization units: method bodies, loop bodies, and traces. They show that method bodies require significantly more code size to capture an equivalent amount of execution time than either traces or loop bodies. This result, together with the properties outlined in Section 1.4, suggest that traces may be a good choice for a unit of compilation.

DynamoRIO Bruening describes a new version of Dynamo which runs on the Intel x86 architecture. The current focus of this work is to provide an efficient environment to instrument real world programs for various purposes such as to improve the security of legacy applications [12, 11].

One interesting application of DynamoRIO was by Sullivan et al [71]. They ran their own tiny interpreter on top of DynamoRIO in the hope that it would be able to dynamically optimize away a significant proportion of interpretation overhead. They did not initially see the results they were hoping for because the indirect dispatch branches confounded Dynamo's trace selection. They responded by creating a small interface by which the interpreter could programatically give DynamoRIO hints about the relationship between the virtual pc and the hardware pc. This was their way around what we call the context problem in Section 3.5. Whereas interpretation slowed down by almost a factor of two using regular DynamoRIO, after they had inserted calls to the hint API, they saw speedups of about 20% on a set of small benchmarks. Baron [8] reports similar performance results running a similarly modified Kaffe

JVM [79].

Last Executed Iteration (LEI)

Hiniker, Hazelwood and Smith performed a simulation study evaluating enhancements to the basic Dynamo trace selection heuristics [40]. They observed two main problems with Dynamo's NET heuristic. The first problem, *trace separation*, occurs when traces that turn out to often execute sequentially happen to be placed far apart in the trace cache, hurting the locality of reference of code in the instruction cache. LEI maintains a branch history mechanism as part of its trace collection system that allows it to do a better job handling loop nests, requiring fewer traces to span the nest. The second problem, excessive code duplication, occurs when many different paths become hot through a region of code. The problem is caused when a trace exit becomes hot and a new trace is generated that diverges from the preexisting trace for only one or a few blocks before rejoining its path. As a consequence the new trace replicates blocks of the old trace from the place they rejoin to their common end. Combining several such observed traces together forms a region with multiple paths and less duplication. A simulation study suggests that using their heuristics, fewer, smaller selected traces will account for the same proportion of execution time.

2.6 Hotpath

Gal, Probst and Franz describe the Hotpath project [32]. Hotpath extends JamVM (one of the interpreters we use for our experiments) to be a trace oriented mixed-mode system. They focus on traces starting at loop headers and do not compile traces other than those in loops. Thus, they do not attempt trace linking as described by Dynamo, but rather "merge" traces that originate from side exits leading back to loop headers. This technique allows Hotpath to compile loop nests. They describe an interesting way of modeling traces using single static assignment (SSA) [21] that exploits the constrained flow of control present in traces. This both

simplifies their construction of SSA and allows very efficient optimization. Their experimental results show excellent speedup, within a factor of two of Sun's HotSpot, for scientific style loop nests like those in the LU, SOR and Linpack benchmarks, and more modest speedup, around a factor of two over interpretation, for FFT. No results are given for tests in the SPECjvm98 suite, perhaps because their system does not yet support "trace merging across (inlined) method invocations" [32, page 151]. The optimization techniques they describe seem complimentary to the overall architecture we propose in Chapter 6.

2.7 Chapter Summary

In this chapter we briefly traced the development of high-level language virtual machines from interpreters to dynamic optimizing compilers. We saw that interpreter designs may perform poorly on modern, highly pipelined processors, because current dispatch mechanisms cause too many branch mispredictions. This will be discussed in more detail in Section 3.5. Later, in Chapter 4, we describe our solution to the problem.

Currently JIT compilers compile entire methods or inlined method nests. Since hot methods may contain cold code this forces the JIT compiler and runtime system to support late binding. Should the cold code later become hot a method-based JIT must recompile the containing method or inlined method nest to optimize the newly hot code. These issues add complexity to a method oriented system that could be avoided if compiled code contained no cold code. The HP Dynamo binary optimizer project defines a suitable candidate for a dynamically identified unit of compilation, namely the hot interprocedural path, or trace.. In Chapter 6 we describe how a virtual machine can compile traces to incrementally compile code as it becomes hot.

Chapter 3

Dispatch Techniques

In this chapter we expand on our discussion of interpretation by examining several dispatch techniques in detail. In Chapter 2 we defined dispatch as the mechanism used by a high level language virtual machine to transfer control from the code to emulate one virtual instruction to the next. This chapter has the flavor of a tutorial as we trace the evolution of dispatch techniques from the simplest to the highest performing.

Although in most cases we will give a small C language example to illustrate the way the interpreter is structured, this should not be understood to mean that all interpreters are hand written C programs. Precisely because so many dispatch mechanisms exist, some researchers argue that the interpreter portion of a virtual machine should be generated from some more generic representation [29, 68].

Section 3.1 describes switch dispatch, the simplest dispatch technique. Section 3.2 introduces call threading, which figures prominently in our work. Section 3.3 describes direct threading, a common technique that suffers from branch misprediction problems. Section 3.4 briefly describes branch prediction resources in modern processors. Section 3.5 defines the *context problem*, our term for the challenge to branch prediction posed by interpretation. Subroutine threading is introduced in Section 3.6. Finally, Section 3.7 describes related work that eliminates dispatch overhead by inlining or replicating virtual instruction bodies.

3.1 Switch Dispatch

Switch dispatch, perhaps the simplest dispatch mechanism, is illustrated by Figure 3.1. Although the persistent representation of a Java class is standards-defined, the representation of a loaded virtual program is up to the VM designer. In this case we show how an interpreter might choose a representation that is less compact than possible for simplicity and speed of interpretation. In the figure, the loaded representation appears on the bottom left. Each virtual opcode is represented as a full word token even though a byte would suffice. Arguments, for those virtual instructions that take them, are also stored in full words following the opcode. This avoids any alignment issues on machines that penalize unaligned loads and stores.

Figure 3.1 illustrates the situation just before the statement `c=a+b+1` is executed. The box on the right of the figure represents the C implementation of the interpreter. The `vPC` points to the word in the loaded representation corresponding to the first instance of `iload`. The interpreter works by executing one iteration of the dispatch loop for each virtual instruction it executes, switching on the token representing each virtual instruction. Each virtual instruction is implemented by a `case` in the `switch` statement. Virtual instruction bodies are simply the compiler-generated code for each case.

Every instance of a virtual instruction consumes at least one word in the internal representation, namely the word occupied by the virtual opcode. Virtual instructions that take operands are longer. This motivates the strategy used to maintain the `vPC`. The dispatch loop always bumps the `vPC` to account for the opcode and bodies that consume operands bump the `vPC` further, one word per operand.

Although no virtual branch instructions are illustrated in the figure, they operate by assigning a new value to the `vPC` for taken branches.

A switch interpreter is relatively slow due to the overhead of the dispatch loop and the switch. Despite this, switch interpreters are commonly used in production (e.g. in the JavaScript and Python interpreters). Presumably this is because switch dispatch can be implemented in ANSI standard C and so it is very portable.

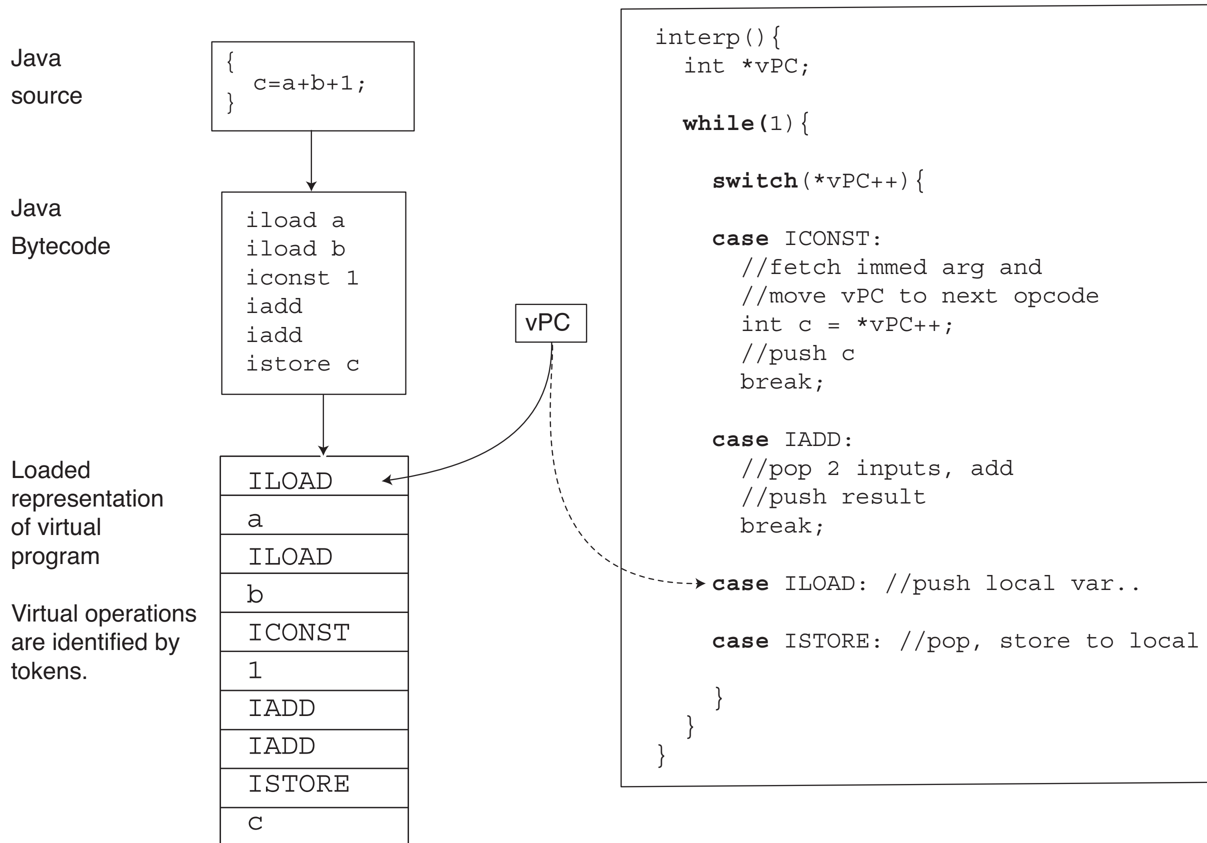


Figure 3.1: A switch interpreter loads each virtual instruction as a virtual opcode, or token, corresponding to the case of the switch statement that implements it. Virtual instructions that take immediate operands, like `iconst`, must fetch them from the `vPC` and adjust the `vPC` past the operand. Virtual instructions which do not need operands, like `iadd`, do not need to adjust the `vPC`.

3.2 Direct Call Threading

Another portable way to organize an interpreter is to write each virtual instruction as a function and dispatch it via a function pointer. Figure 3.2 shows each virtual instruction body implemented as a C function. While the loaded representation used by the switch interpreter represents the opcode of each virtual instruction as a token, direct call threading represents each virtual opcode as the address of the function that implements it. Thus, by treating the `vPC` as a function pointer, a direct call-threaded interpreter can execute each instruction in turn.

In the figure, the `vPC` is a static variable which means the `interp` function as shown is not re-entrant. Our example aims only to convey the flavor of call threading. In Chapter 6 we will show how a more complex approach to direct call threading can perform about as well as switch threading.

A variation of this technique is described by Ertl [26]. For historical reasons the name "direct" is given to interpreters which store the *address* of the virtual instruction bodies in the loaded representation. Presumably this is because they can "directly" obtain the address of a body, rather than using a mapping table (or switch statement) to convert a virtual opcode to the address of the body. However, the name can be confusing as the actual machine instructions used for dispatch are indirect branches. (In this case, an *indirect* call).

Next we will describe direct threading, perhaps the most well-known high performance dispatch technique.

3.3 Direct Threading

Like in direct call threading, a virtual program is loaded into a direct-threaded interpreter as a list of body addresses and operands. We will refer to the list as the *Direct Threading Table*, or DTT, and refer to locations in the DTT as *slots*.

Interpretation begins by initializing the `vPC` to the first slot in the DTT, and then jumping to the address stored there. A direct-threaded interpreter does not need a dispatch loop like

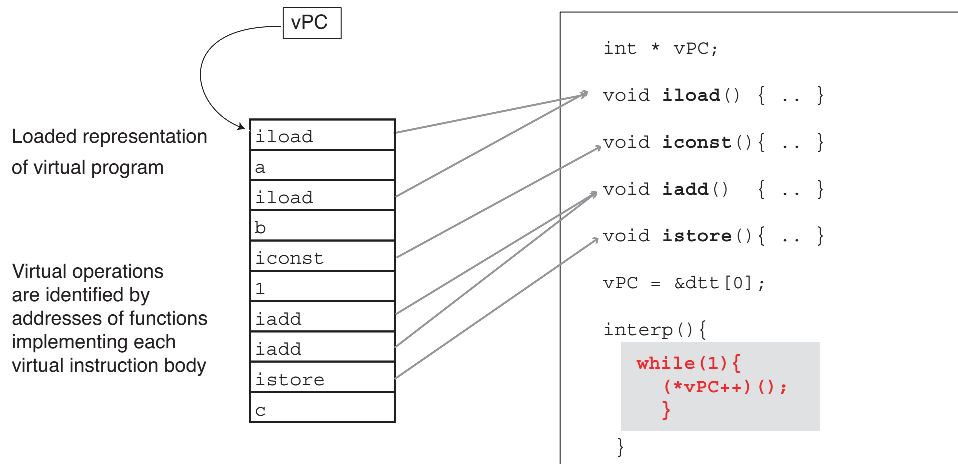


Figure 3.2: A direct call-threaded interpreter packages each virtual instruction body as a function. The shaded box highlights the dispatch loop showing how virtual instructions are dispatched through a function pointer. Direct call threading requires the loaded representation of the program to point to the *address* of the function implementing each virtual instruction.

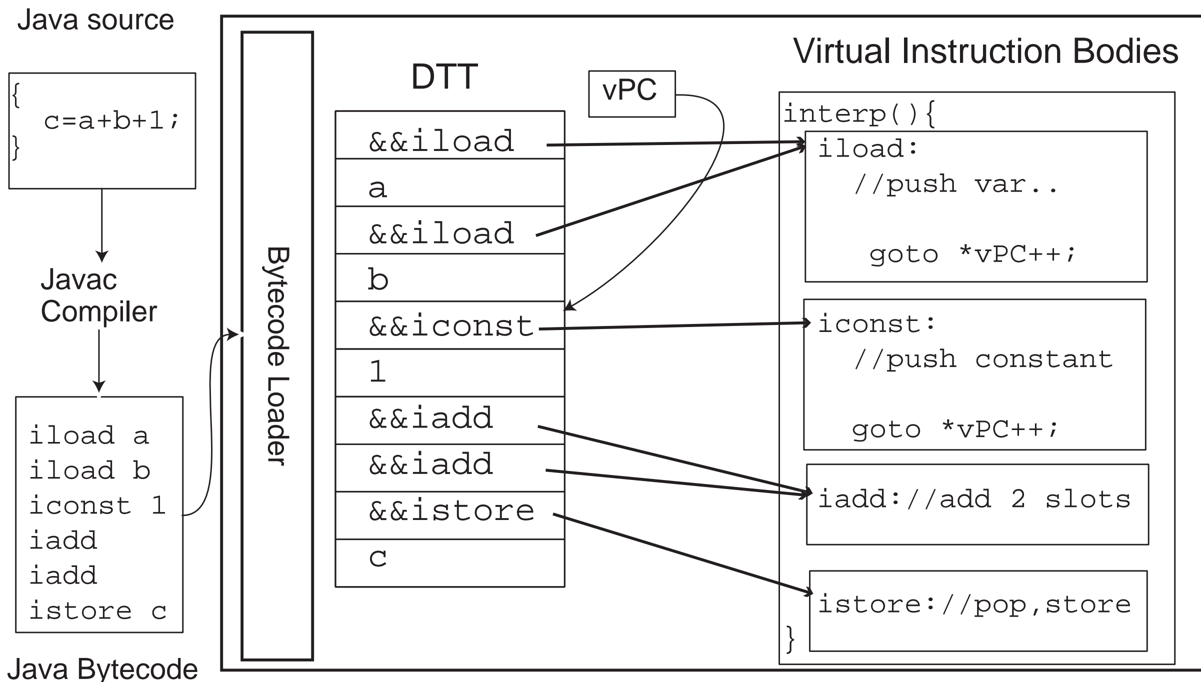


Figure 3.3: Direct-threaded Interpreter showing how Java Source code compiled to Java bytecode is loaded into the Direct Threading Table (DTT). The virtual instruction bodies are written in a single C function, each identified by a separate label. The double-ampersand (&&) shown in the DTT is gcc syntax for the address of a label.

(a) Pentium IV assembly	(b) Power PC assembly
`mov %eax = (%rx) ;` *rx is vPC*	`lwz r2 = 0(rx)`
`addl 4,%rx`	`mtctr r2`
`jmp (%eax)`	`addi rx,rx,4`
	`bctr`

Figure 3.4: Machine instructions used for direct dispatch. On both platforms assume that some general purpose register, `rx`, has been dedicated for the `vPC`. Note that on the PowerPC indirect branches are two part instructions that first load the `ctr` register and then branch to its contents.

direct call threading or switch dispatch. Instead, as can be seen in Figure 3.3, each body ends with `goto *vPC++`, which transfers control to the next instruction.

In C, bodies are identified by a label. Common C language extensions permit the address of this label to be taken, which is used when initializing the DTT. GNU's `gcc`, as well as C compilers produced by Intel, IBM and Sun Microsystems all support the label-as-value and computed goto extensions, making direct threading quite portable.

Direct threading requires fewer instructions and is faster than direct call threading or switch dispatch. Assembler for the dispatch sequence is shown in Figure 3.4. When executing the indirect branch in Figure 3.4(a) the Pentium IV will speculatively dispatch instructions using a predicted target address. The PowerPC uses a different strategy for indirect branches, as shown in Figure 3.4(b). First the target address is loaded into a register, and then a branch is executed to this register address. Rather than speculate, the PowerPC stalls until the target address is known, although other instructions may be scheduled between the load and the branch (like the `addi` in Figure 3.4) to reduce or eliminate these stalls.

3.4 Dynamic Hardware Branch Prediction

There is a rich body of research on branch prediction, since branches are otherwise very costly on pipelined architectures. In this thesis we care only about techniques adopted by real micro-processors.

The primary mechanism used to predict indirect branches on modern computers is the

branch target buffer (BTB). The BTB is a hardware table in the CPU that associates the destination of a small set of branches with their address [39]. The idea is to simply remember the previous destination of each branch. This is the same as assuming that the destination of each indirect branch is correlated with the address in memory of the branch instruction itself.

The Pentium IV implements a 4K entry BTB [41]. (There is no mention of a BTB in the PowerPC 970 programmers manual [45].) Direct threading confounds the BTB because all instances of a given virtual instruction compete for the same BTB slot.

Another kind of dynamic branch predictor is used for conditional branch instructions. Conditional branches are relative, or direct, branches so there are only two possible destinations. The challenge lies in predicting whether the branch will be taken or fall through. For this purpose modern processors implement a *branch history table*. The PowerPC 7410, as an example, deploys a 2048 entry 2 bit branch history table [53]. Direct threading also confounds the branch history table as all the instances of each conditional branch virtual instruction compete for the same branch history table entry. In this case the hard to predict branch is not an explicit dispatch branch but rather the result of an `if` statement in a virtual branch instruction body. This will be discussed in more detail in Section 4.3.

Return instructions can be predicted perfectly using a stack of addresses pushed by call instructions. The Pentium IV has a 16 entry *return address stack* [41] whereas the PPC970 uses a similar structure called the *link stack* [45].

3.5 The Context Problem

Mispredicted branches pose a serious challenge to modern processors because they threaten to starve the processor of instructions. The problem is that before the destination of the branch is known the execution of the pipeline may run dry. To perform at full speed, modern CPU's need to keep their pipelines full by correctly predicting branch targets.

Ertl points out that the assumptions underlying the design of indirect branch predictors are

usually wrong for direct-threaded interpreters [27, 28]. In a direct-threaded interpreter, there is only *one* indirect jump instruction per virtual instruction. For example, in the fragment of virtual code illustrated in Figure 2.1, there are two instances of `iload` followed by an instance of `iconst`. The indirect dispatch branch at the end of the `iload` body will execute twice. The first time, in the context of the first instance of `iload,` it will branch back to the entry point of the the `iload` body, whereas in the context of the second `iload` it will branch to `iconst`. Thus, the hardware will likely mispredict the second execution of the dispatch branch.

The performance impact of this can be hard to predict. For instance, if a tight loop in a virtual program happens to contain a sequence of unique virtual instructions, the BTB may successfully predict each one. On the other hand, if the sequence contains duplicate virtual instructions, the BTB may mispredict all of them.

This problem is even worse for direct call threading and switch dispatch. For these techniques there is only one dispatch branch and so all dispatches share the same BTB entry. Direct call threading will mispredict all dispatches except when the same virtual instruction body is dispatched multiple times consecutively.

Another perspective is that the destination of the indirect dispatch branch is unpredictable because its destination is not correlated with the hardware `pc`. Instead, its destination is correlated to the `vPC`. We refer to this lack of correlation between the hardware `pc` and `vPC` as the *context problem*. We choose the term *context* following its use in *context sensitive inlining* [38] because in both cases the context of shared code (in their case methods, in our case virtual instruction bodies) is important to consider.

3.6 Subroutine Threading

Forth is organized as a collection of callable bodies of code called *words*. Words can be user defined or built into the system. Meaningful Forth words are composed of built-in and user-

defined words and execute by dispatching their constituent words in turn. A Forth implementation is said to be *subroutine-threaded* if a word is compiled to a sequence of *native call instructions,* one call for each constituent word. Since a built-in Forth word is loosely analogous to a callable virtual instruction body we could conceivably use subroutine threading in any high level language virtual machine that implements virtual instruction bodies as callable. In such a system the loaded representation of a virtual method would include a sequence of native call instructions, one to dispatch each virtual instruction in the virtual method.

Curley [20, 19] describes a subroutine-threaded Forth for the 68000 CPU. He improves the resulting code by inlining small opcode bodies, and converts virtual branch opcodes to single native branch instructions. He credits Charles Moore, the inventor of Forth, with discovering these ideas much earlier. Outside of Forth, there is little thorough literature on subroutine threading. In particular, few authors address the problem of where to store virtual instruction operands. In Section 4.2, we document how operands are handled in our implementation of subroutine threading.

The choice of optimal dispatch technique depends on the hardware platform, because dispatch is highly dependent on micro-architectural features. On earlier hardware, *call* and *return* were both expensive and hence subroutine threading required two costly branches, versus one in the case of direct threading. Rodriguez [61] presents the trade offs for various dispatch types on several 8 and 16-bit CPUs. For example, he finds direct threading is faster than subroutine threading on a 6809 CPU, because the `jsr` and `ret` instruction require extra cycles to push and pop the return address stack. On the other hand, Curley found subroutine threading faster on the 68000 [19]. On modern hardware the cost of the return is much lower, due to return branch prediction hardware, while the cost of direct threading has increased due to misprediction. In Chapter 5 we quantify this effect on a few modern CPUs.

3.7 Optimizing Dispatch

Much of the work on interpreters has focused on how to optimize dispatch. In general dispatch optimizations can be divided into two broad classes: those which refine the dispatch itself, and those which alter the bodies so that they are more efficient or simply require fewer dispatches. Switch dispatch and direct threading belong to the first class, as does subroutine threading. Kogge [49] remains a definitive description of many threaded code dispatch techniques. Next, we will discuss superinstruction formation and replication, which are in the second class.

3.7.1 Superinstructions

Superinstructions reduce the number of dispatches. Consider the code to add a constant integer to a variable. This may require loading the variable onto the expression stack, loading the constant, adding, and storing back to the variable. VM designers can instead extend the virtual instruction set with a single superinstruction that performs the work of all four virtual instructions. This technique is limited, however, because the virtual instruction encoding (often one byte per opcode) may allow only a limited number of instructions, and the number of desirable superinstructions grows large in the number of subsumed atomic instructions. Furthermore, the optimal superinstruction set may change based on the workload. One approach uses profile-feedback to select and create the superinstructions statically (when the interpreter is compiled [29]).

3.7.2 Selective Inlining

Piumarta [59] presents *selective inlining*. Selective inlining constructs superinstructions when the virtual program is loaded. They are created in a relatively portable way, by `memcpy`'ing the compiled code in the bodies, again using GNU C labels-as-values. The idea is to construct (new) super instruction bodies by concatenating the virtual bodies of the virtual instructions that make them up. This works only when the code in the virtual bodies is *position independent*,

meaning that the destination of any relative branch in a body remain in that body. Typically this excludes bodies making C function calls. This technique was first documented earlier [63], but Piumarta's independent discovery inspired many other projects to exploit selective inlining. Like us, he applied his optimization to OCaml, and reports significant speedup on several micro benchmarks. As we discuss in Section 5.3, our technique is separate from, but supports and indeed facilitates, inlining optimizations.

Languages, like Java, that require runtime binding complicate the implementation of selective inlining significantly because at load time little is known about the arguments of many virtual instructions. When a Java method is first loaded some arguments are left unresolved. For instance, the argument of an `invokevirtual` instruction will initially be a string naming the callee. The argument will be re-written the first time the virtual instruction executes to point to a descriptor of the now resolved callee. At the same time, the virtual opcode is rewritten so that subsequently a "quick" form of the virtual instruction body will be dispatched. In Java, if resolution fails, the instruction throws an exception and is not rewritten. The process of rewriting the arguments, and especially the need to point to a new virtual instruction body, complicates superinstruction formation. Gagnon describes a technique that deals with this additional complexity which he implemented in SableVM [31].

Selective inlining requires that the superinstruction starts at a virtual basic block, and ends at or before the end of the block. Ertl's *dynamic superinstructions* [28] also use `memcpy`, but are applied to effect a simple native compilation by inlining bodies for nearly every virtual instruction. Ertl shows how to avoid the basic block constraints, so dispatch to interpreter code is only required for virtual branches and unrelocatable bodies. Vitale and Abdelrahman describe a technique called catenation, which patches Sparc native code so that all implementations can be moved, specializes operands, and converts virtual branches to native, thereby eliminating the virtual program counter [75].

3.7.3 Replication

Replication — creating multiple copies of the opcode body—decreases the number of contexts in which it is executed, and hence increases the chances of successfully predicting the successor [28]. Replication combined with inlining opcode bodies reduces the number of dispatches, and therefore, the average dispatch overhead [59]. In the extreme, one could create a copy for each instruction, eliminating misprediction entirely. This technique results in significant code growth, which may [75] or may not [28] cause cache misses.

3.8 Chapter Summary

In summary, branch mispredictions caused by the context problem limit the performance of a direct-threaded interpreter on a modern processor. We have described several recent dispatch optimization techniques. Some of the techniques improve performance of each dispatch by reducing the number of contexts in which a body is executed. Others reduce the number of dispatches, possibly to zero.

In the next chapter we will describe a new technique for interpretation that deals with the context problem. Our technique, context threading, performs well compared to the interpretation techniques we have described in this chapter.

Chapter 4

Design and Implementation of Efficient Interpretation

This chapter will describe how to efficiently implement an interpreter that calls its virtual instruction bodies. This investigation was motivated by the suggestion we made in Chapter 1, namely that such an interpreter will be easier to extend with a JIT than an interpreter that is direct-threaded or uses switch dispatch. Before tackling the design of our mixed-mode system we need to ensure that the interpreter is efficient.

An obvious, but slow, way to use callable virtual instruction bodies is to build a direct call threaded (DCT) interpreter (see Section 3.2 for a detailed description of the technique.) In a DCT interpreter all bodies are dispatched by the same *indirect* call instruction. The destination of the indirect call is data driven (i.e. by the sequence of virtual instructions that make up the virtual program) and thus impossible for the hardware to predict. As a result, a DCT interpreter suffers a branch misprediction for almost every dispatch.

The main realization driving our approach is that to call each body without misprediction dispatch branches must be *direct* call instructions. Since these can only be generated when virtual instructions are loaded, we generate them ourselves. At load time, each straight-line section of virtual instructions is translated to a sequence of direct call native instructions, each

dispatching the corresponding virtual instruction body. The loaded program is run by jumping to the beginning of the generated sequence of native code, which then emulates the virtual program by calling each virtual instruction body in turn. This approach is very similar to a Forth compile-time technique called subroutine threading, described in Section 3.6.

Subroutine threading dispatches straight-line sequences of virtual instructions very efficiently because no branch mispredictions occur. The generated direct calls pose no prediction challenge because each has only one explicit destination. The destination of the return ending each body is perfectly predicted by the return branch predictor stack implemented by modern processors. In the next chapter we present data showing that subroutine threading runs the SPECjvm98 suite about 20% faster than direct threading.

Subroutine threading handles straight-line virtual code efficiently, but does nothing to improve the dispatch of virtual branch instructions. We introduce *context threading*, which, by generating more sophisticated code for virtual branch instructions, eliminates the branch mispredictions caused by the dispatch of virtual branch instructions as well. Context threading improves the performance of the SPECjvm98 suite by about another 5% over subroutine threading.

Generating and dispatching native code obviously makes our implementation of subroutine threading less portable than many dispatch techniques. However, since subroutine threading requires the generation of only one type of machine instruction, a direct call, its hardware dependency is isolated to a few lines of code. Context threading requires much more machine dependent code generation.

In Chapter 6 we will describe another way of handling virtual branches that requires less complex, less machine dependent code generation, but requires additional runtime infrastructure to identify hot runtime interprocedural paths, or traces.

Although direct-threaded interpreters are known to have poor branch prediction properties, they are also known to have a small instruction cache footprint [62]. Since both branch mispredictions and instruction cache misses are major pipeline hazards, we would like to retain the

good cache behavior of direct-threaded interpreters while improving the branch behavior. Subroutine threading minimally affects code size. This is in contrast to techniques like selective inlining, described in Section 3.7, which improve branch prediction by replicating entire bodies, in effect trading instruction cache size for better branch prediction. In Chapter 7 we will report data showing that subroutine threading causes very few additional stall cycles caused by instruction cache misses as compared to direct threading.

In Section 4.1 we discuss the challenge of virtual branch instructions in general terms. In Section 4.2 we show how to replace straight-line dispatch with subroutine threading. In Section 4.3 we show how to inline conditional and indirect jumps and in Section 4.4 we discuss handling virtual calls and returns with native calls and returns.

4.1 Understanding Branches

Before describing our design, we start with two observations. First, a virtual program will typically contain several types of control flow: conditional and unconditional branches, indirect branches, and calls and returns. We must also consider the dispatch of straight-line virtual instructions. For direct-threaded interpreters, straight-line execution is just as expensive as handling virtual branches, since *all* virtual instructions are dispatched with an indirect branch. Second, the dynamic execution path of the virtual program will contain patterns (loops, for example) that are similar in nature to the patterns found when executing native code. These control flow patterns originate in the algorithm that the virtual program implements.

As described in Section 3.4, modern microprocessors have considerable resources devoted to identifying these patterns in native code, and exploiting them to predict branches. Direct threading uses only indirect branches for dispatch and, due to the context problem, the patterns that exist in the virtual program are largely hidden from the microprocessor.

The spirit of our approach is to expose these virtual control flow patterns to the hardware, such that the physical execution path matches the virtual execution path. To achieve this goal,

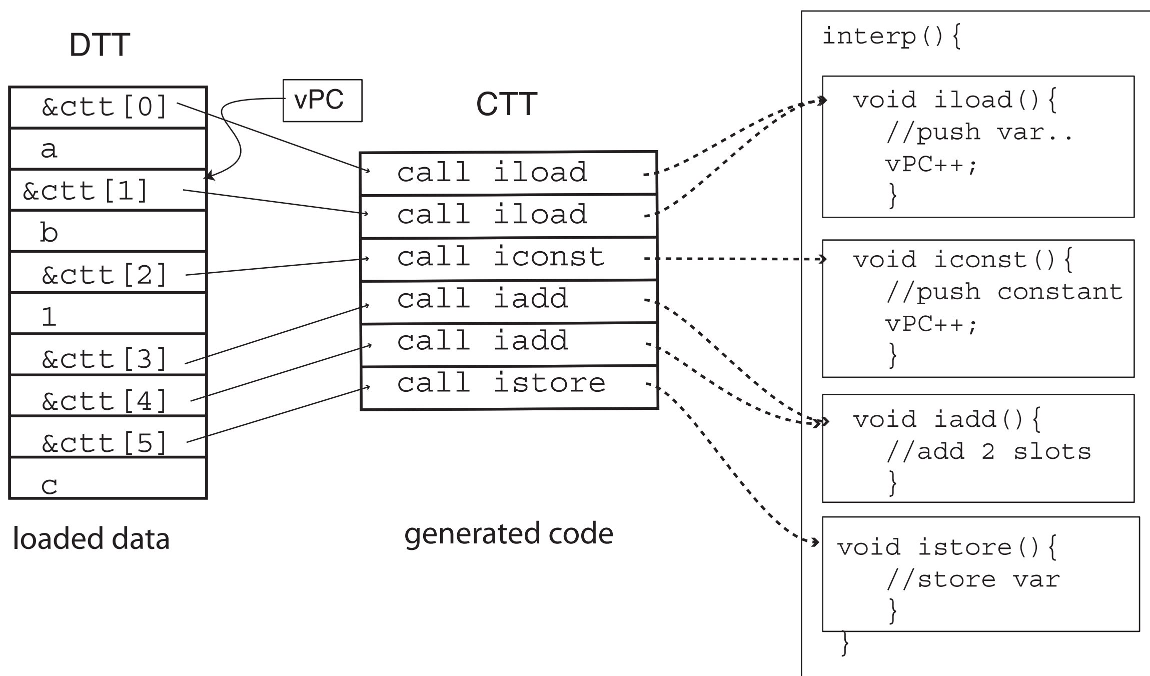


Figure 4.1: Subroutine Threaded Interpreter showing how the CTT contains one generated direct call instruction for each virtual instruction and how the first entry in the DTT corresponding to each virtual instruction points to generated code to dispatch it. Callable bodies are shown here as nested functions for illustration only.

we generate dispatch code at load time that enables the different types of hardware prediction resources to predict the different types of virtual control flow transfers. We strive to maintain the property that the virtual program counter is precisely correlated with the physical program counter and in fact, when all our techniques are combined, there is a one-to-one mapping between them at most control flow points.

4.2 Handling Linear Dispatch

The dispatch of straight-line virtual instructions is the largest single source of branches when executing an interpreter. Any technique that hopes to improve branch prediction accuracy must address straight-line dispatch. An obvious solution is inlining, as it eliminates the dispatch entirely for straight-line sequences of virtual instructions. However, as mentioned in Section 3.7, the increase in code size caused by aggressive inlining has the potential to overwhelm the

```
interp(){

  iload:
     //push local var

     asm ("ret");
     goto *vPC++;

  iconst:
     //push constant

     asm ("ret");
     goto *vPC++;

}
```

Figure 4.2: Direct threaded bodies retrofitted as callable routines by inserting inline assembler return instructions. This example is for Pentium 4 and hence ends each body with a `ret` instruction. The `asm` statement is an extension to the C language, inline assembler, provided by gcc and many other compilers.

benefits with the cost of increased instruction cache misses [75].

Rather than eliminate dispatch, we propose an alternative organization for the interpreter in which native call and return instructions are used. This approach is conceptually elegant because the subroutine is a natural unit of abstraction to express the implementation of virtual instruction bodies.

Figure 4.1 illustrates our implementation of subroutine threading, using the same example program as Figure 3.3. In this case, we show the state of the virtual machine *after* the first virtual instruction has been executed. We add a new structure to the interpreter architecture, called the *Context Threading Table* (CTT), which contains a sequence of native call instructions. Each native call dispatches the body for its virtual instruction. Although Figure 4.1 shows each body as a nested function, in fact we implement this by ending each non-branching opcode body with a native return instruction as shown in Figure 4.2.

The handling of immediate arguments to virtual instructions is perhaps the biggest difference between our implementation of subroutine threading and the approach used by Forth. Forth words pop all their arguments from the expression stack — there is no concept of an im-

mediate operand. Thus, there is no need for a structure like the DTT. The virtual instruction set defined by a Java virtual machine includes many instructions which take immediate operands. Hence, in Java, we need both the direct threading table (DTT) and the CTT. In Section 3.3 we described how the DTT is used to store immediate operands, and to correctly resolve virtual control transfer instructions. In direct threading, entries in the DTT point to virtual instruction bodies, whereas in subroutine threading they refer to call sites in the CTT.

It may seem counterintuitive to improve dispatch performance by calling each body because the latency of a call and return may be greater than an indirect jump. This is not the real issue. On modern microprocessors the extra cost of the call (if any) is far outweighed by the benefit of eliminating a large source of unpredictable branches, as the data presented in the next chapter will show.

4.3 Handling Virtual Branches

Subroutine threading handles the branches that implement the dispatch of straight-line virtual instructions; however, the control flow of the virtual program is still hidden from the hardware. That is, bodies that perform virtual branches still have no context. There are two problems, the first relating to shared indirect branch prediction resources, and the second relating to a lack of history context for conditional branch prediction resources.

Figure 4.3 introduces a new Java example, this time including a virtual branch. Consider the implementation of `ifeq`, shaded in the figure. Prediction of the indirect branch at "(a)" may be problematic, because *all* instances of `ifeq` instructions in the virtual program share the same indirect branch instruction (and hence have a single prediction context).

Figure 4.4 illustrates *branch replication*, a simple solution to the first of these problems. The idea is to generate an indirect branch instruction in the CTT immediately following the dispatch of the virtual branch. Virtual branch bodies have been modified to end with a native return instruction and the only result of dispatching a branch body is the side effect of setting

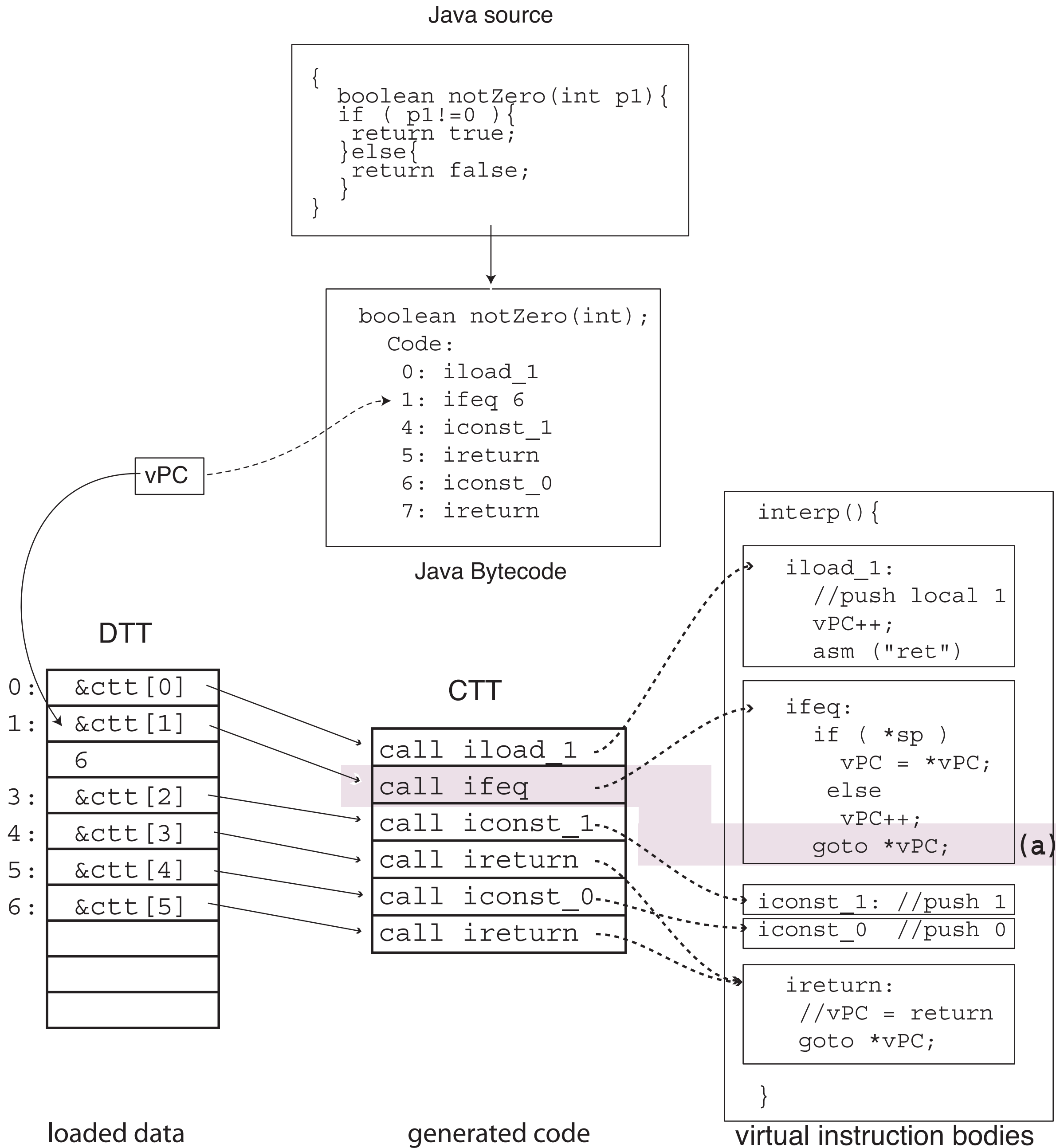


Figure 4.3: Subroutine Threading does not not address branch instructions. Unlike straight line virtual instructions, virtual branch bodies end with an indirect branch, just like direct threading. (Note: When a body is called the vPC always points to the slot in the DTT corresponding to its first argument, or, if there are no operands, to the following instruction.)

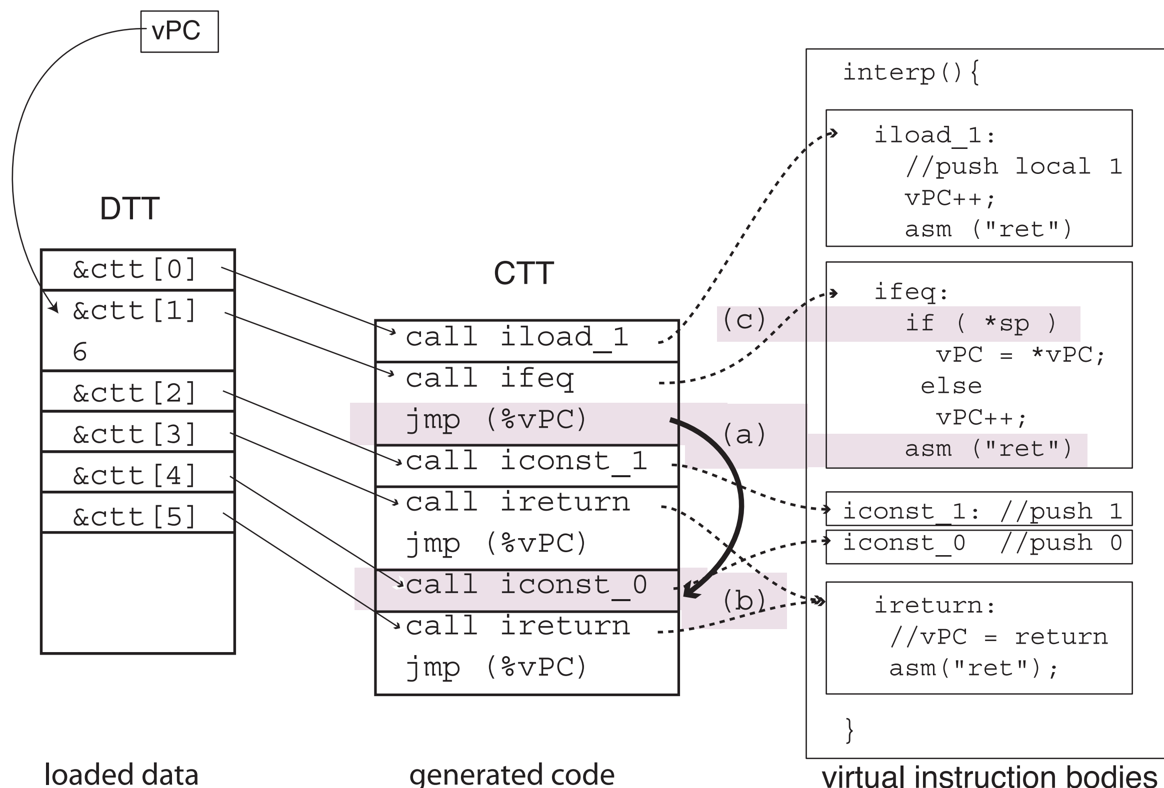


Figure 4.4: Context threading with branch replication illustrating the "replicated" indirect branch (a) in the CTT. The fact that the indirect branch corresponds to only one virtual instruction gives it better prediction context. The heavy arrow from (a) to (b) is followed when the virtual branch is taken. Prediction problems remain in the code compiled from the `if` statement labelled (c)

the `vPC` to the destination. The result is that each virtual branch instruction has its own indirect branch predictor entry. Branch replication is an appropriate term because the indirect branch ending the branch body has been copied to potentially many places in the CTT.)

Branch replication is attractive because it is simple and produces the desired context with a minimum of new generated instructions. However, it has a number of drawbacks. First, for branching opcodes, we execute three hardware control transfers (a call to the body, a return, and the replicated indirect branch), which is an unnecessary overhead. Second, we still use the overly general indirect branch instruction, even in cases like `goto` where we would prefer a simpler direct native branch. Third, by only replicating the dispatch part of the virtual instruction, we do not take full advantage of the conditional branch predictor resources provided by

the hardware. This is because the `if` statement in the body, marked (c) in the figure, is shared by all instances of `ifeq`. Due to these limitations, we only use branch replication for indirect virtual branches and exceptions[1]

Branch inlining, illustrated by Figure 4.5, is a technique that generates code for the bodies of virtual branch instructions into the CTT. In the figure we show how our system inlines the `ifeq` instruction. The generated native code, shaded in the figure, implements the same if-then-else logic as the original direct-threaded virtual instruction body. The inlined conditional branch instruction (`jne`, “(a)” in the figure) is thus fully exposed to the Pentium’s conditional branch prediction hardware.

On the Pentium, branch inlining reduces pressure on the branch target buffer, or BTB, since conditional branches use the conditional branch predictors instead. The virtual conditional branches now appear as real conditional branches to the hardware. The dispatch of the body has been entirely eliminated.

The primary cost of branch inlining is increased code size, but this is modest because, at least for languages like Java and OCaml, virtual branch instructions are simple and have small bodies. For instance, on the Pentium IV, most branch instructions can be inlined with no more than 10 words, at worst a few additional i-cache lines.

The obvious challenge of branch inlining, apart from the hard labor required to implement it, is that the generated code is not portable and assumes detailed knowledge of the virtual bodies it must interoperate with.

4.4 Handling Virtual Call and Return

The only significant source of control transfers that remain in the virtual program is virtual method invocation and return. For successful branch prediction, the real problem is not the virtual call, which has only a few possible destinations, but rather the virtual return, which

[1] OCaml defines explicit exception virtual instructions

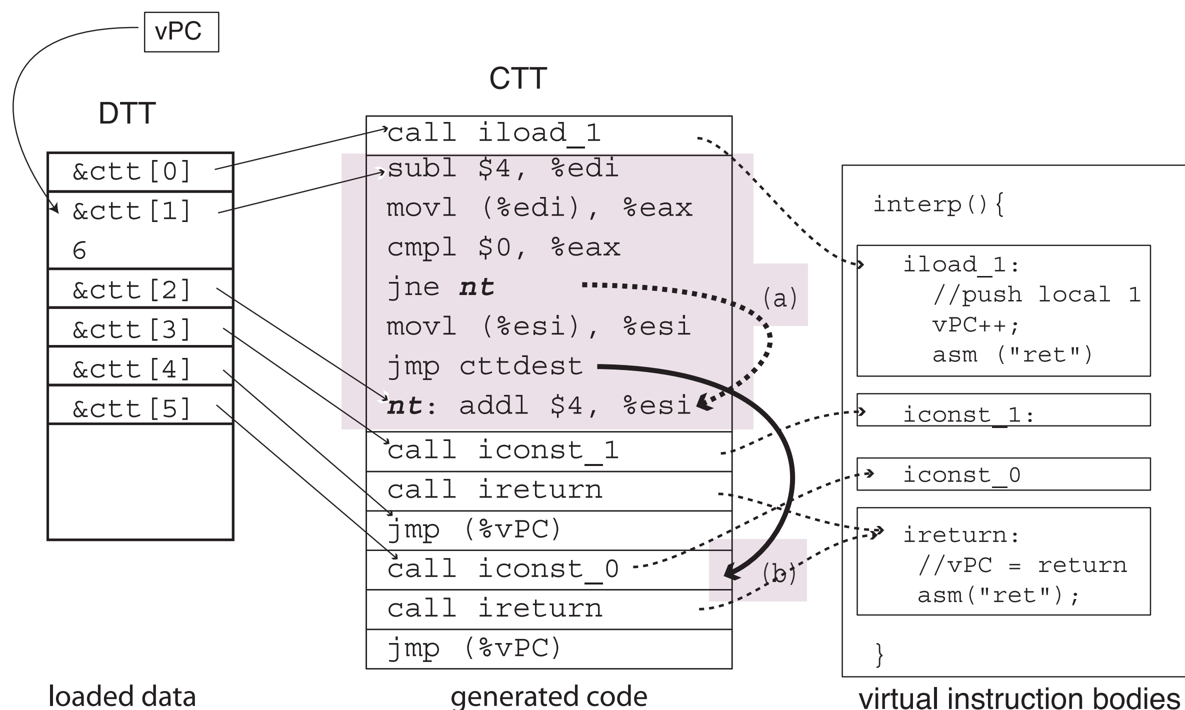


Figure 4.5: Context-threaded VM Interpreter: Branch Inlining. The dashed arrow (a) illustrates the inlined conditional branch instruction, now fully exposed to the branch prediction hardware, and the heavy arrow (b) illustrates a direct branch implementing the not taken path. The generated code (shaded) assumes the `vPC` is in register `esi` and the Java expression stack pointer is in register `edi`. (In reality, we dedicate registers in the way shown for SableVM on the PowerPC only. On the Pentium4, due to lack of registers, the `vPC` is actually stored on the stack.)

potentially has many destinations, one for each callsite of the method. As noted previously, the hardware already has an elegant solution to this problem in the form of the return address stack. We need only to deploy this resource to predict virtual returns.

We describe our solution with reference to Figure 4.6. The virtual method invocation body, Java's `invokestatic` in the figure, must transfer control to the first virtual instruction of the callee. Our goal is to generate dispatch code so that the corresponding virtual return instruction makes use of the hardware's return branch predictors.

We begin at the virtual call instruction (just before label "(a)" in the figure). The body of the `invokestatic` creates a new frame for the callee, and then sets the `vPC` to the entry point of the callee ("(b)" in the figure) before returning back to the CTT. Similar to branch replication,

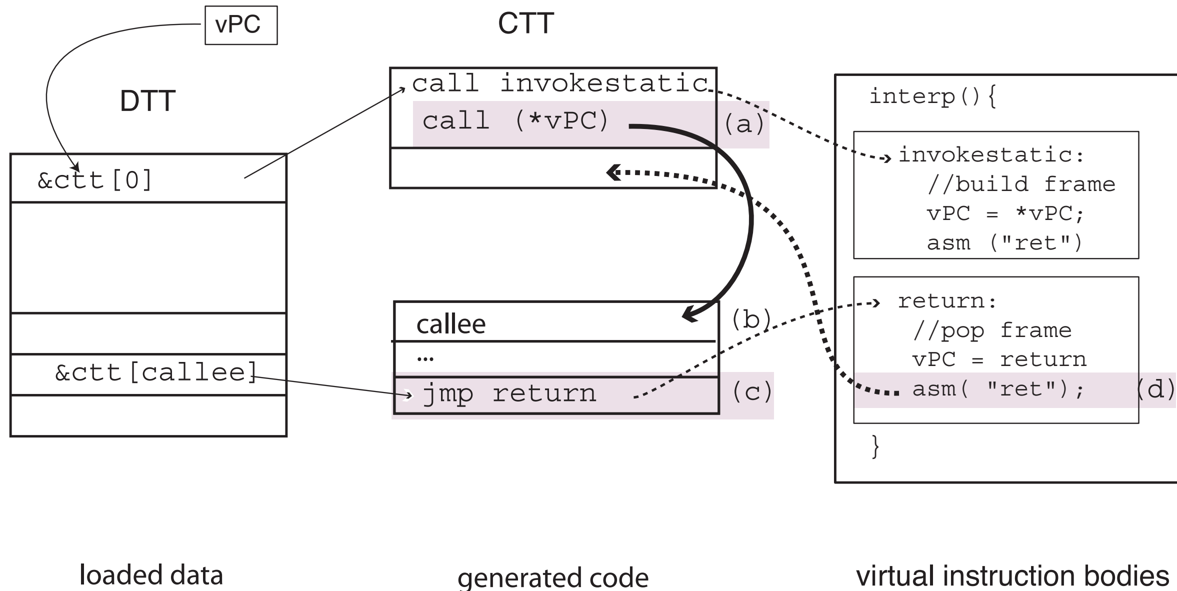


Figure 4.6: Context Threading Apply-Return Inlining on Pentium. The generated code *calls* the `invokestatic` virtual instruction body but *jumps* (instruction at (c) is a `jmp`) to the return body.

we insert a new native *call indirect* instruction following “(a)” in the CTT to transfer control to the start of the callee, shown as a solid arrow from “(a)” to “(b)” in the figure. The call indirect has the desired side effect of pushing CTT location (a) onto the hardware’s return address stack. The first instruction of the callee is then dispatched. At the end of the callee, we modify the virtual return instruction as follows. In the CTT, at “(c)”, we emit a native direct *jump*, an x86 `jmp` in the figure, to dispatch the body of the virtual return. This direct branch avoids perturbing the return address stack. The body of the virtual return now returns all the way back to the instruction following the original virtual call. This is shown as the dotted arrow from “(d)” to following “(a)”. We refer to this technique as *apply/return inlining*[2].

With this final step, we have a complete technique that aligns all virtual program control flow with the corresponding native flow. There are however, some practical challenges to implementing our design for apply/return inlining. First, one must take care to match the hardware stack against the virtual program stack. For instance, in OCaml, exceptions unwind

[2]“apply” is the name of the (generalized) function call opcode in OCaml where we first implemented the technique.

the virtual machine stack; the hardware stack must be unwound in a corresponding manner. Second, some runtime environments are extremely sensitive to hardware stack manipulations, since they use or modify the machine stack pointer for their own purposes. In such cases, it is possible to create a separate stack structure and swap between the two at virtual invocation and return points. This approach would introduce significant overhead, and is only justified if apply/return inlining provides a substantial performance benefit.

4.5 Chapter Summary

The code generation described in this chapter is carried out when each virtual method is loaded. The idea is to generate relatively simple code that exposes the dispatch branch instructions to the hardware branch predictors of the processor.

In the next chapter we present data showing that our approach is effective in the sense that branch mispredictions are reduced and performance is improved. Subroutine threading is by far the most effective, especially when its relatively simplicity and small amount of machine dependent code are taken into account. Branch inlining is the most complicated and least portable.

Our implementation of context threading has at least two potential problems. First, effort is expended at load time for regions of code that may never execute. This could penalize performance when large amounts of cold code are present. Second, is it awkward to interpose profiling instrumentation around the virtual instruction bodies dispatched from the CTT. The difficulty stems from the fact that subroutine threading, like direct threading, does not need a dispatch loop. This means that calls to profiling code must be generated in amongst the generated dispatch code in the CTT. Removing instrumentation after it is needed requires much code rewriting. The resulting system, though efficient, is fragile and hard to work with [80].

In Chapter 6 we describe a different approach to efficient interpretation that addresses these two problems. There, we describe a different approach that generates simple code for hot inter-

procedural paths, or traces. This allows us to exploit the efficacy and simplicity of subroutine threading for straight-line code at the same time as eliminate the mispredictions caused by virtual branch instructions.

Chapter 5

Evaluation of Context Threading

In this chapter we evaluate context threading by comparing its performance to direct threading and direct-threaded selective inlining. We evaluate the impact of each of our techniques on Pentium 4 and PowerPC by measuring the performance of a modified version of SableVM, a Java virtual machine and ocamlrun, an OCaml interpreter. We explore the differences between context threading and SableVM's selective inlining further by measuring a simple extension of context threading we call tiny inlining. Finally, we investigate the limitations of our techniques by comparing the performance improvement of subroutine-threaded Tcl and subroutine-threaded OCaml to direct threading on Sparc.

The overall results show that dispatching virtual instructions by calling virtual instructions bodies is very effective for Java and OCaml on Pentium IV and PowerPC platforms. In fact, subroutine threading outperforms direct threading by a healthy margin of about 20%. Context threading is almost as fast as selective inlining as implemented by SableVM. Since these are dispatch optimizations, they offer performance benefits depending on the proportion of dispatch to real work. Thus, when a Tcl interpreter is modified to be subroutine-threaded, performance increases much less than OCaml on the same Sparc processor, only about 5%.

We begin by describing our experimental setup in Section 5.1. We investigate how effectively our techniques address pipeline branch hazards in Section 5.2.1, and the overall effect

on execution time in Section 5.2.2. Section 5.3 demonstrates that context threading is complementary to inlining and results in performance comparable to SableVM's implementation of selective inlining. Finally, Section 5.4 discusses a few of the limitations of context threading by studying the performance of Vitale's subroutine-threaded Tcl [76, Figure 1] and OCaml, on Sparc.

5.1 Experimental Set-up

We evaluate our techniques by modifying interpreters for Java and OCaml to run on Pentium IV, PowerPC 7410 and PPC970. The Pentium and PowerPC are processors used by PC and Macintosh workstations and many types of servers, so performance on these platforms is relevant. The Pentium and PowerPC provide different architectures for indirect branches (Figure 3.4 illustrates the differences) so we ensure our techniques work for both approaches.

Our experimental approach is to evaluate performance by measuring elapsed time. This is simple to measure and always relevant. We guard against intermittent events polluting any single run by always averaging across three executions of each benchmark.

We report pipeline hazards using the performance measurement counters of each processor. These vary widely not only between the Pentium and the PowerPC but also within each family. This is a challenge on the PowerPC, where IBM's modern PowerPC 970 is a desirable processor to measure, but has no performance counters for stalls caused by indirect branches. Thus, we use an older processor model, the PowerPC 7410, because it implements performance counters that the PowerPC 970 does not.

5.1.1 Virtual Machines and Benchmarks

We choose two virtual machines for our experiments. OCaml is a simple, very cleanly implemented interpreter. However, there is only one implementation to measure and only a few relatively small benchmark programs are available. For this reason we also modified SableVM,

Table 5.1: Description of OCaml benchmarks. Raw elapsed time and branch hazard data for direct-threaded runs.

		Pentium IV		PowerPC 7410		PPC970	Lines
Benchmark	Description	Time (TSC*10^8)	Branch Mispredicts (MPT*10^6)	Time (Cycles*10^8)	Branch Stalls (Cycles*10^6)	Elapsed Time (sec)	of Source Code
boyer	Boyer theorem prover	3.34	7.21	1.8	43.9	0.18	903
fft	Fast Fourier transform	31.9	52.0	18.1	506	1.43	187
fib	Fibonacci by recursion	2.12	3.03	2.0	64.7	0.19	23
genlex	A lexer generator	1.90	3.62	1.6	27.1	0.11	2682
kb	A knowledge base program	17.9	42.9	9.5	283	0.96	611
nucleic	nucleic acid's structure	14.3	19.9	95.2	2660	6.24	3231
quicksort	Quicksort	9.94	20.1	7.2	264	0.70	91
sieve	Sieve of Eratosthenes	3.04	1.90	2.7	39.0	0.16	55
soli	A classic peg game	7.00	16.2	4.0	158	0.47	110
takc	Takeuchi function (curried)	4.25	7.66	3.3	114	0.33	22
taku	Takeuchi function (tuplified)	7.24	15.7	5.1	183	0.52	21

a Java Virtual Machine.

OCaml We chose OCaml as representative of a class of efficient, stack-based interpreters that use direct-threaded dispatch. The bytecode bodies of the interpreter, in C, have been hand-tuned extensively, to the point of using gcc inline assembler extensions to hand-allocate important variables to dedicated registers. The implementation of the OCaml interpreter is clean and easy to modify [13, 1].

OCaml Benchmarks The benchmarks in Table 5.1 make up the standard OCaml benchmark suite[1]. `Boyer`, `kb`, `quicksort` and `sieve` do mostly integer processing, while `nucleic` and `fft` are mostly floating point benchmarks. `Soli` is an exhaustive search algorithm that solves a solitaire peg game. `Fib`, `taku`, and `takc` are tiny, highly-recursive programs which calculate integer values.

`Fib`, `taku`, and `takc` are unusual because they contain very few distinct virtual instructions, and in some cases use only one instance of each. This has two important consequences. First, the indirect branch in direct-threaded dispatch is relatively predictable. Second, even minor changes can have dramatic effects (both positive and negative) because so few instructions

[1] `ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/benchmarks/objcaml.tar.gz`

Table 5.2: Description of SPECjvm98 Java benchmarks. Raw elapsed time and branch hazard data for direct-threaded runs.

		Pentium IV		PowerPC 7410		PPC970
Benchmark	Description	Time (TSC*10^{11})	Branch Mispredicts (MPT*10^{9})	Time (Cycles*10^{10})	Branch Stalls (Cycles*10^{8})	Elapsed Time (sec)
compress	Modified Lempel-Ziv compression	4.48	7.13	17.0	493	127.7
db	performs multiple database functions	1.96	2.05	7.5	240	65.1
jack	A Java parser generator	0.71	0.65	2.7	67	18.9
javac	the Java compiler from the JDK 1.0.2	1.59	1.43	6.1	160	44.7
jess	Java Expert Shell System	1.04	1.12	4.2	110	29.8
mpegaudio	decompresses MPEG Layer-3 audio files	3.72	5.70	14.0	460	106.0
mtrt	two thread variant of raytrace	1.06	1.04	5.3	120	26.8
raytrace	a raytracer rendering	1.00	1.03	5.2	120	31.2
scimark	performs FFT SOR and LU, 'large'	4.40	6.32	18.0	690	118.1
soot	java bytecode to bytecode optimizer	1.09	1.05	2.7	71	35.5

contribute to the behavior.

SableVM SableVM is a Java Virtual Machine built for quick interpretation. SableVM implements multiple dispatch mechanisms, including switch, direct threading, and selective inlining (which SableVM calls *inline threading* [31]). The support for multiple dispatch mechanisms facilitated our work to add context threading and allows for comparisons against other techniques, like inlining, that also address branch mispredictions. Finally, as part of its own inlining infrastructure SableVM builds tables describing which virtual instruction bodies can be safely inlined using memcpy. This made our tiny inlining implementation very simple.

Java Benchmarks SableVM experiments were run on the complete SPECjvm98 [65] suite (`compress`, `db`, `mpegaudio`, `raytrace`, `mtrt`, `jack`, `jess` and `javac`), one large object-oriented application (`soot` [74]) and one scientific application (`scimark` [60]). Table 7.1 summarizes the key characteristics of these benchmarks.

5.1.2 Performance and Pipeline Hazard Measurements

On both platforms we measure elapsed time averaged over three runs to mitigate noise caused by intermittent system events. We necessarily use platform and operating systems dependent methods to estimate pipeline hazards.

Pentium IV Measurements The Pentium IV (P4) processor speculatively dispatches instructions based on branch predictions. As discussed in Section 3.5, the indirect branches used for direct-threaded dispatch are often mispredicted due to the lack of context. Ideally, we could measure the cycles the processor stalls due to mispredictions of these branches, but the P4 does not provide a performance counter for this purpose. Instead, we count the number of *mispredicted taken branches* (MPT) to measure how our techniques effect branch prediction. We measure time on the P4 with the cycle-accurate *time stamp counter* (TSC) register. We count both MPT and TSC events using our own Linux kernel module, which collects complete data for the multithreaded Java benchmarks[2].

PowerPC Measurements We need to characterize the cost of branches differently on the PowerPC than on the P4, as the PPC does not speculate on indirect branches. Instead, split branches are used (as shown in Figure 3.4(b)) and the PPC stalls until the branch destination is known. Hence, we would like to count the number of cycles stalled due to link and count register dependencies. Unfortunately, PPC970 chips do not provide a performance counter for this purpose; however, the older PPC7410 CPU has a counter (counter 15, “stall on LR/CTR dependency”) that provides exactly the information we need [53]. On the PPC7410, we also use the hardware counters to obtain overall execution times in terms of clock cycles. We expect that the branch stall penalty should be larger on more deeply-pipelined CPUs like the PPC970, however, we cannot directly verify this. Instead, we report only elapsed execution time for the PPC970.

5.2 Interpreting the data

[2]MPT events are counted with performance counter 8 by setting the P4 CCCR to 0x0003b000 and the ESCR to value 0xc001004 [46]

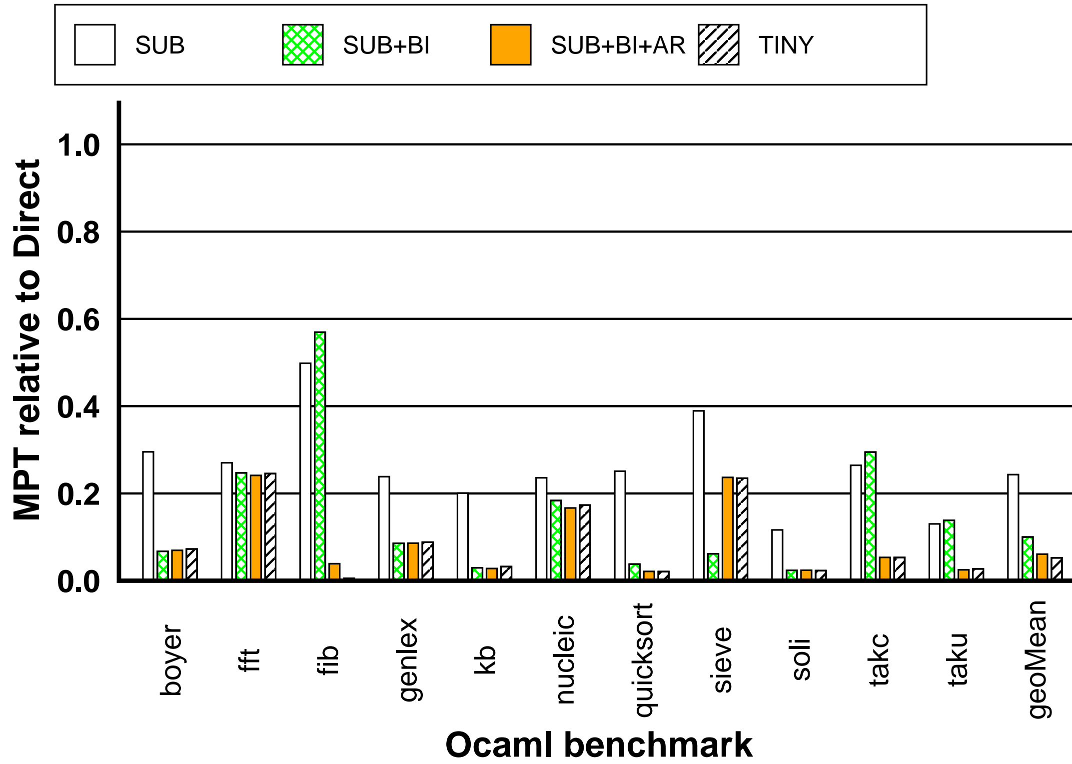
Table 5.3: (a) Guide to Technique description.

Technique	Key	Description
Subroutine Threading	SUB	Section 4.2
Branch Inlining	SUB+BI	Section 4.3
Context Threading	SUB+BI+AR	Section 4.4
Tiny Inlining	TINY	Section 5.3
Selective Inlining (sablevm)	SABLEVM	Section 3.7

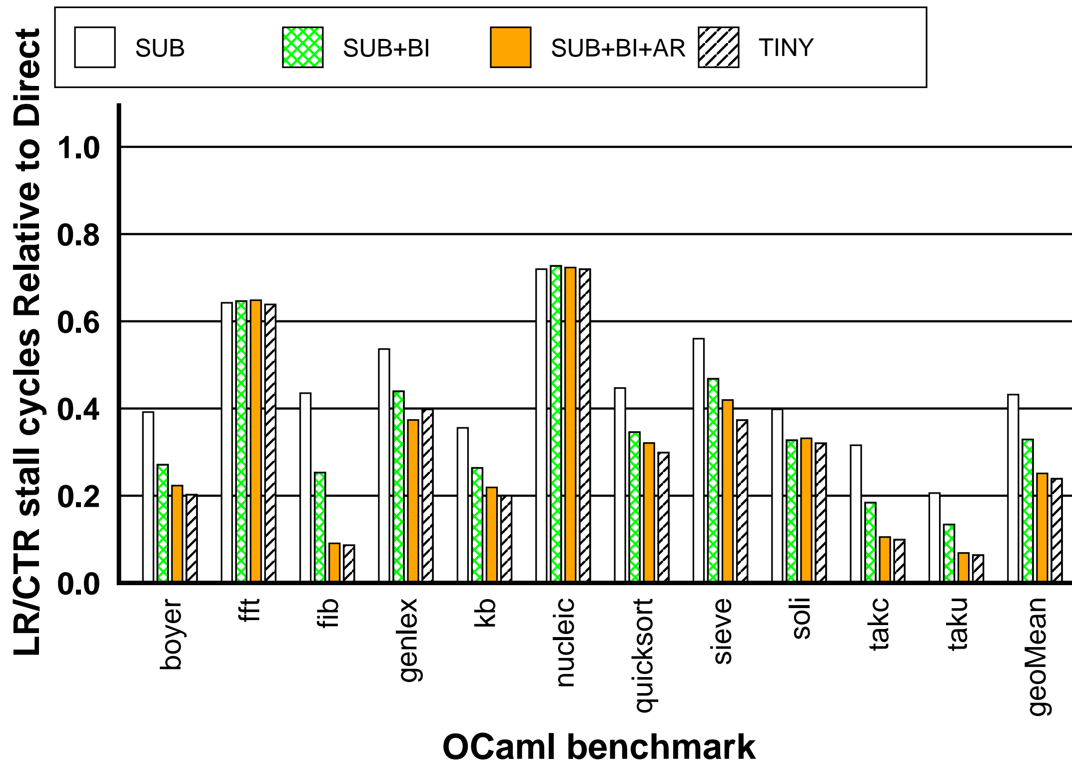
(b) Guide to performance data figures.

Interpreter	Hazards	P4/PPC7410 Performance	PPC970 time
OCaml	Figure 5.1 on the facing page	Figure 5.3 on page 69	Figure 5.5 (a) on page 71
Java (SableVM)	Figure 5.2 on page 66	Figure 5.4 on page 70	Figure 5.5 (b) on page 71

In presenting our results, we normalize all experiments to the direct threading case, since it is considered a state-of-the art dispatch technique. (The source distributions of both OCaml and SableVM configure for direct threading.) We give the absolute execution times and branch hazard statistics for each benchmark and platform using direct threading in Tables 5.1 and 7.1. Bar graphs in the following sections show the contributions of each component of our technique: subroutine threading only (labeled SUB); subroutine threading plus branch inlining and branch replication for exceptions and indirect branches (labeled SUB+BI); and our complete context threading implementation which includes apply/return inlining (labeled SUB+BI+AR. We include bars for selective inlining in SableVM (labeled SABLEVM) and our own simple inlining technique (labeled TINY) to facilitate comparisons, although inlining results are not discussed until Section 5.3. We do not show a bar for direct threading because it would, by definition, have height 1.0. Table 5.3 provides a key to the acronyms used as labels in the following graphs.

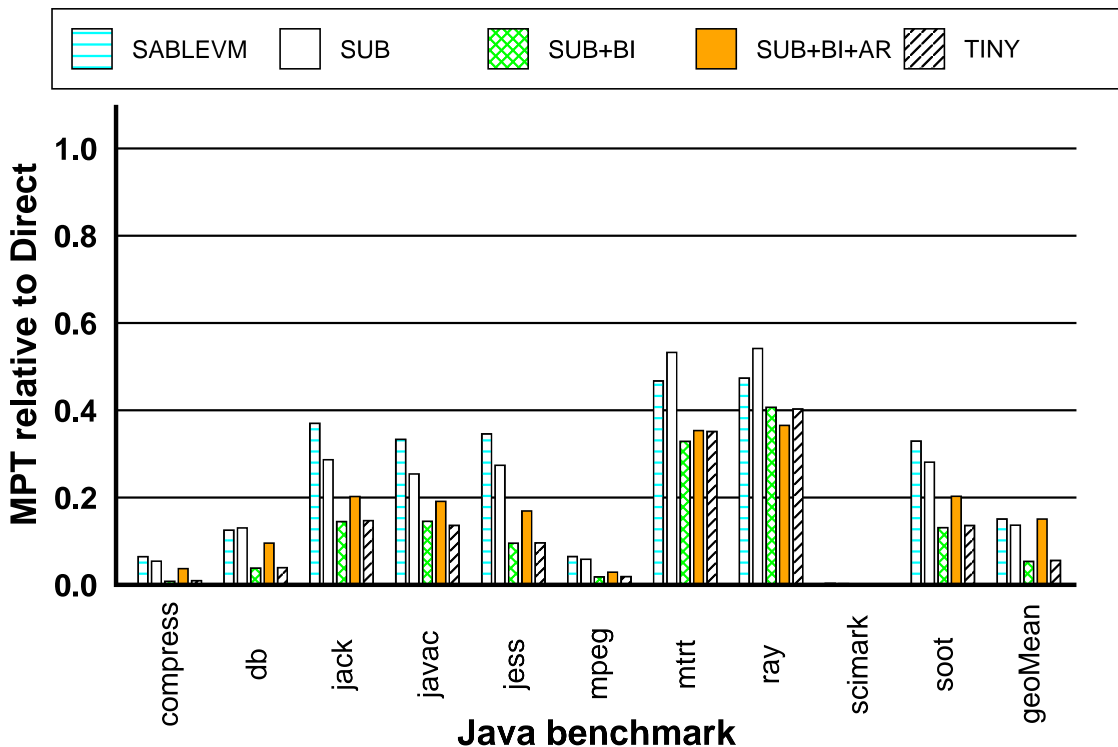


(a) Pentium 4 Mispredicted Taken Branches

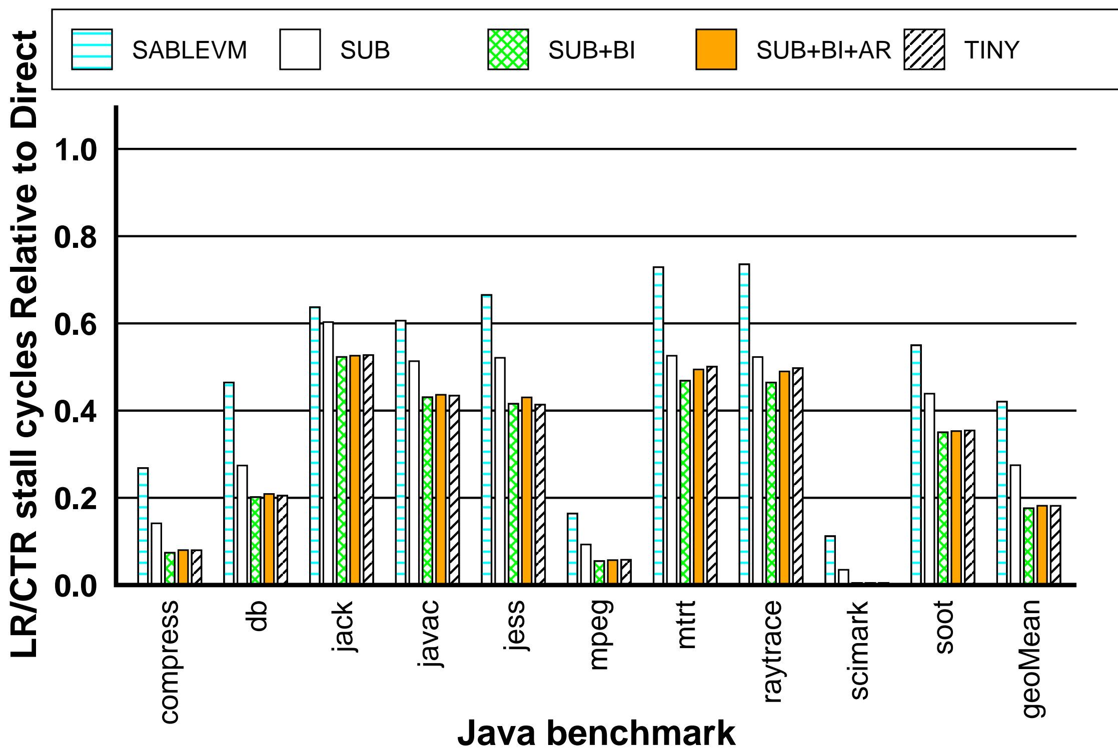


(b) PPC 7410 LR/CTR stall cycles

Figure 5.1: OCaml Pipeline Hazards Relative to Direct Threading



(a) Pentium 4 Mispredicted Taken Branches



(b) PPC7410 - LR/CTR stall cycles

Figure 5.2: Java Pipeline Hazards Relative to Direct Threading

5.2.1 Effect on Pipeline Branch Hazards

Context threading was designed to align virtual program state with physical machine state to improve branch prediction and reduce pipeline branch hazards. We begin our evaluation by examining how well we have met this goal.

Figure 5.1 reports the extent to which context threading reduces pipeline branch hazards for the OCaml benchmarks, while Figure 5.2 reports these results for the Java benchmarks on SableVM. On the top of both figures, the graph labeled (a) presents the results on the P4, where we count mispredicted taken branches (MPT). On bottom, graphs labeled (b) present the effect on LR/CTR stall cycles on the PPC7410. The last cluster of each bar graph reports the geometric mean across all benchmarks.

Context threading eliminates most of the mispredicted taken branches (MPT) on the Pentium IV and LR/CTR stall cycles on the PPC7410, with similar overall effects for both interpreters. Examining Figures 5.1 and 5.2 reveals that subroutine threading has the single greatest impact, reducing MPT by an average of 75% for OCaml and 85% for SableVM on the P4, and reducing LR/CTR stalls by 60% and 75% on average for the PPC7410. This result matches our expectations because subroutine threading addresses the largest single source of unpredictable branches—the dispatch used for straight-line sequences of virtual instructions. Branch inlining has the next largest effect, since conditional branches are the most significant remaining pipeline hazard after applying subroutine threading. On the P4, branch inlining cuts the remaining MPTs by about 60%. On the PPC7410 branch inlining has a smaller, though still important effect, eliminating about 25% of the remaining LR/CTR stall cycles. A notable exception to the MPT trend occurs for the OCaml micro-benchmarks `Fib`, `takc` and `taku`. These tiny recursive micro benchmarks contain few duplicate virtual instructions and so the Pentium's BTB mostly predicts correctly. Hence, inlining the conditional branches cannot help.

Interestingly, the same three OCaml micro benchmarks `Fib`, `takc` and `taku` that challenge branch inlining on the P4 also reap the greatest benefit from apply/return inlining, as shown in Figure 5.1(a). (This appears as the significant improvement of SUB+BI+AR relative

to SUB+BI.) Due to the recursive nature of these benchmarks, their performance is dominated by the behavior of virtual calls and returns. Thus, we expect predicting the returns to have significant impact.

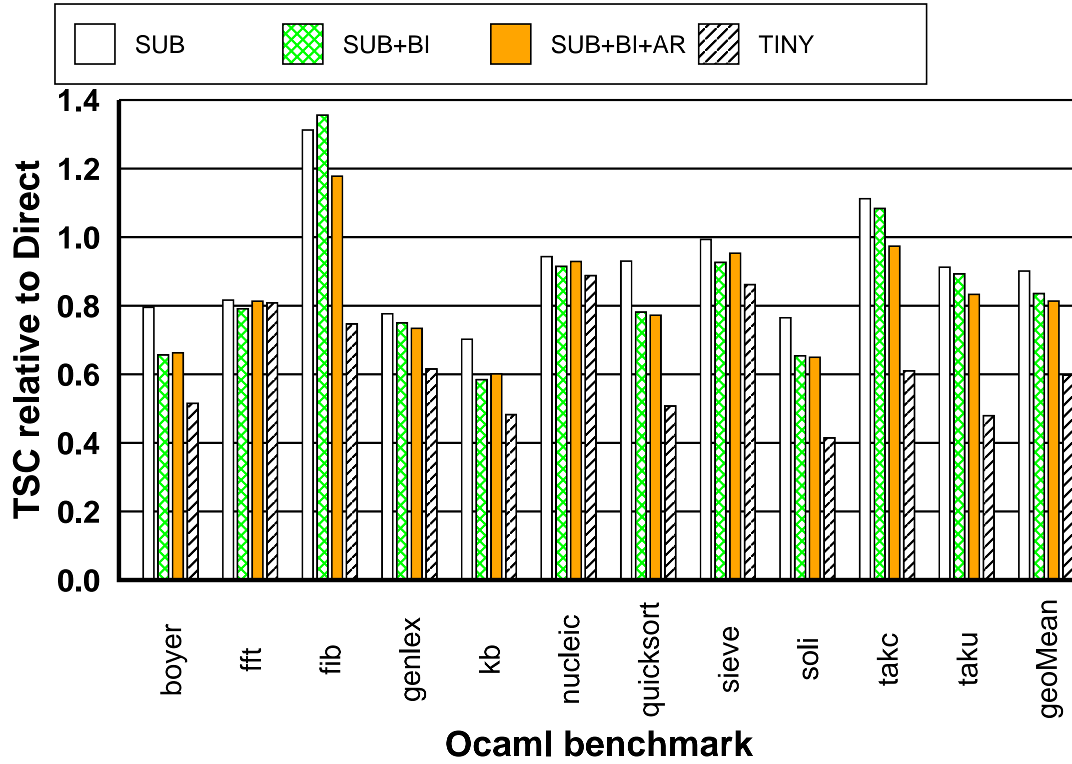
For SableVM on the P4, however, our implementation of apply/return inlining is restricted by the fact that gcc-generated code touches the processor's `esp` register. Rather than implement a complicated stack switching technique, as discussed in Section 4.4, we allow the virtual and machine stacks to become misaligned when SableVM manipulates the `esp` directly. This increases the overhead of our apply/return inlining implementation, presumably by reducing the effectiveness of the return address stack predictor. On the PPC7410, the effect of apply/return inlining on LR/CTR stalls is very small for SableVM.

Having shown that our techniques can significantly reduce pipeline branch hazards, we now examine the impact of these reductions on overall execution time.

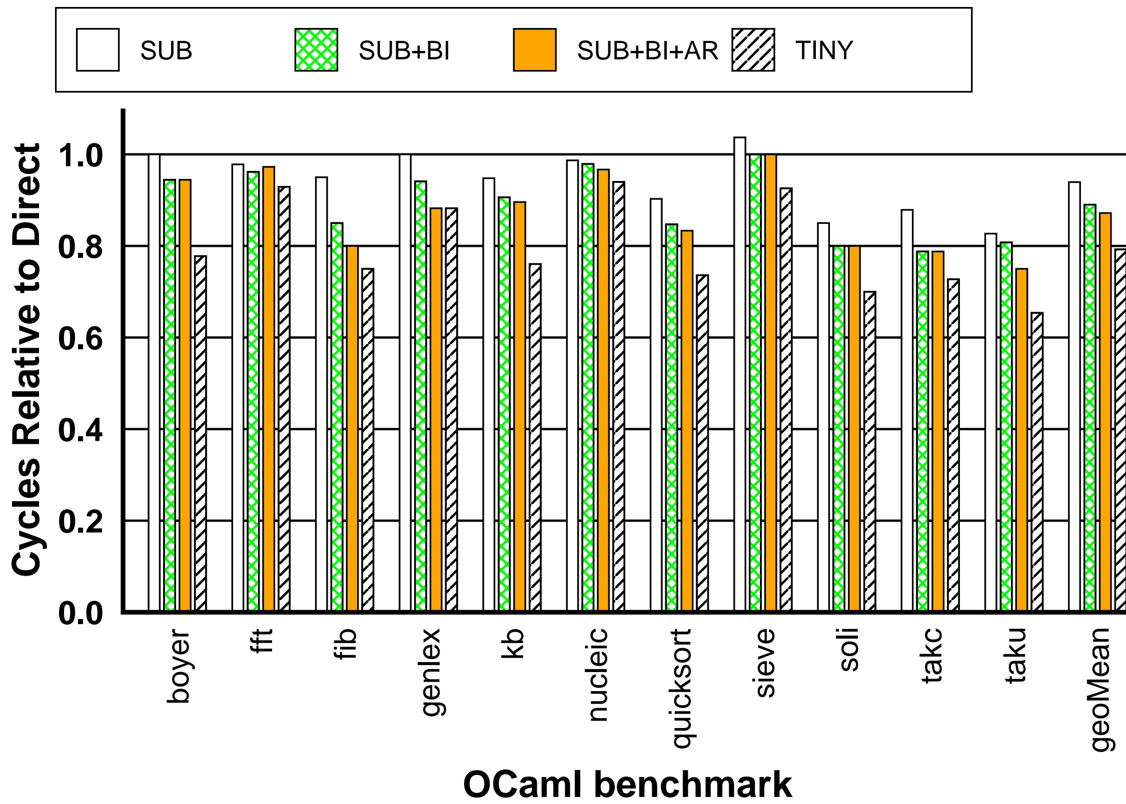
5.2.2 Performance

Context threading improves branch prediction, resulting in better use of the pipelines on both the P4 and the PPC. However, using a native *call/return* pair for each dispatch increases instruction overhead. In this section, we examine the net result of these two effects on overall execution time. As before, all data is reported relative to direct threading.

Figures 5.3 and 5.4 show results for the OCaml and SableVM benchmarks respectively. They are organized in the same way as the previous section, with P4 results on the top, labeled (a), and PPC7410 results on bottom, labeled (b). Figure 5.5 reports the performance of OCaml and SableVM on the PPC970 CPU. The geometric means (rightmost cluster) in Figures 5.3, 5.4 and 5.5 show that context threading significantly outperforms direct threading on both virtual machines and on all three architectures. The geometric mean execution time of the

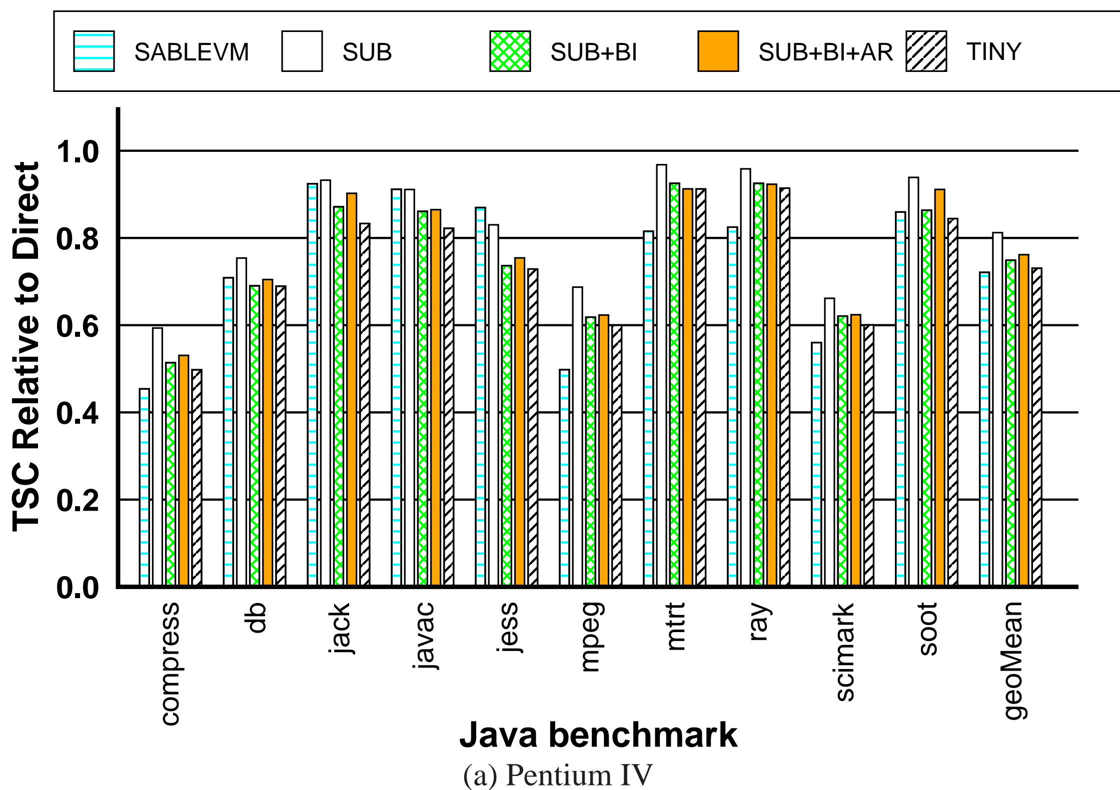


(a) Pentium 4

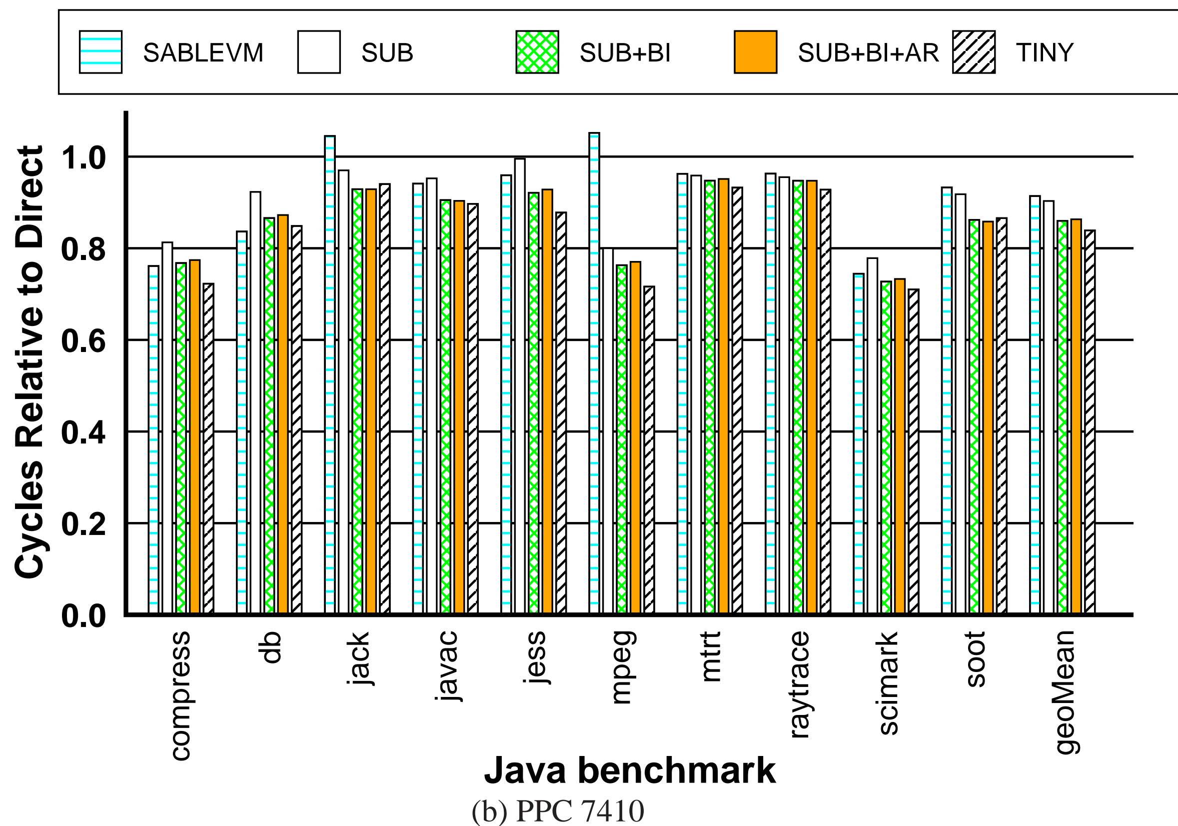


(b) PPC7410

Figure 5.3: OCaml Elapsed Time Relative to Direct Threading

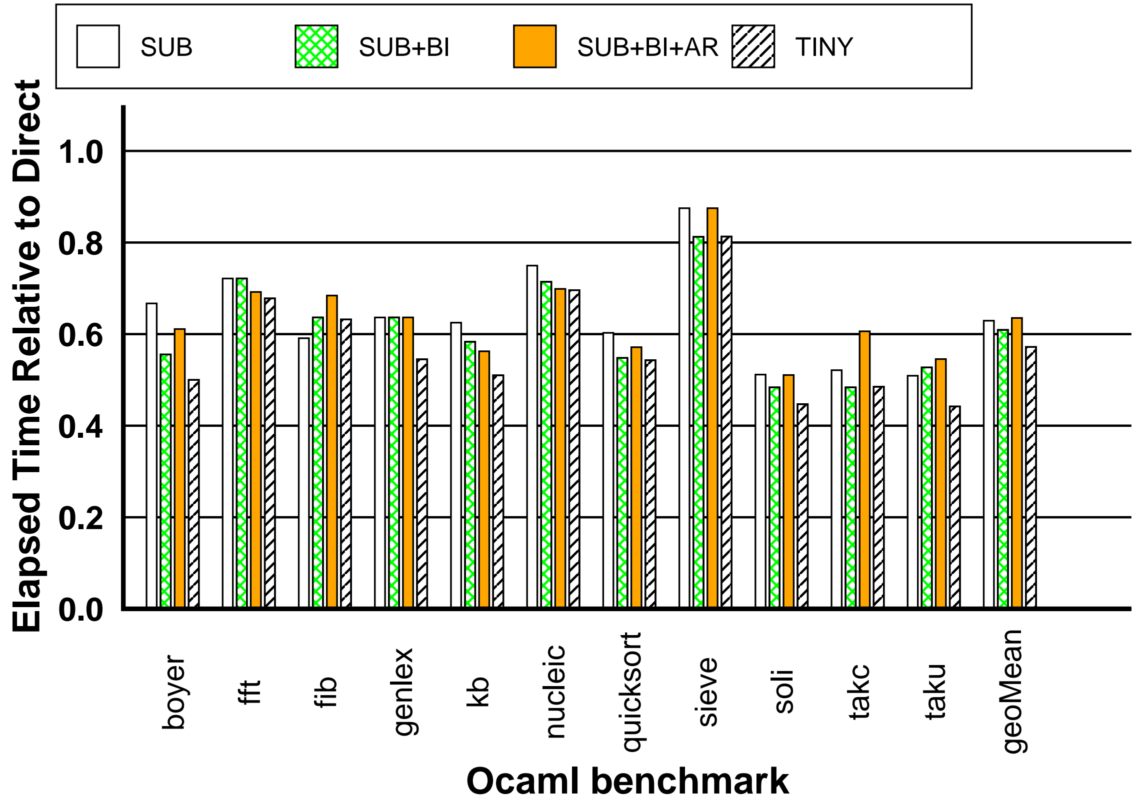


(a) Pentium IV



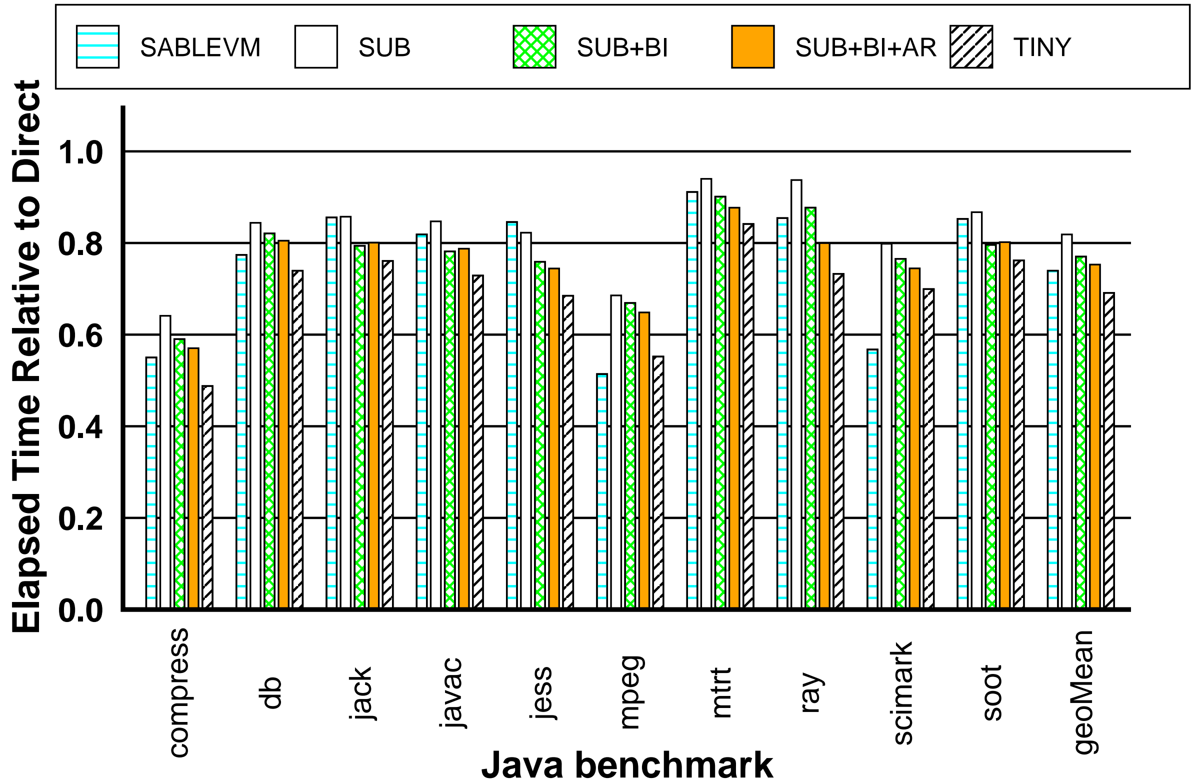
(b) PPC 7410

Figure 5.4: SableVM Elapsed Time Relative to Direct Threading



s

(a) OCaml PPC970 elapsed (real) seconds



(b) SableVM PPC970 elapsed (real) seconds

Figure 5.5: PPC970 Elapsed Time Relative to Direct Threading

OCaml VM is about 19% lower for context threading than direct threading on P4, 9% lower on PPC7410, and 39% lower on the PPC970. For SableVM, SUB+BI+AR, compared with direct threading, runs about 17% faster on the PPC7410 and 26% faster on both the P4 and PPC970. Although we cannot measure the cost of LR/CTR stalls on the PPC970, the greater reductions in execution time are consistent with its more deeply-pipelined design (23 stages vs. 7 for the PPC7410).

Across interpreters and architectures, the effect of our techniques is clear. Subroutine threading has the single largest impact on elapsed time. Branch inlining has the next largest impact eliminating an additional 3–7% of the elapsed time. In general, the reductions in execution time track the reductions in branch hazards seen in Figures 5.1 and 5.2. The longer path length of our dispatch technique are most evident in the OCaml benchmarks `fib` and `takc` on the P4 where the improvements in branch prediction (relative to direct threading) are minor. These tiny benchmarks compile into unique instances of a few virtual instructions. This means that there is little or no sharing of BTB slots between instances and hence fewer mispredictions.

The effect of apply/return inlining on execution time is minimal overall, changing the geometric mean by only $\pm 1\%$ with no discernible pattern. Given the limited performance benefit and added complexity, a general deployment of apply/return inlining does not seem worthwhile. Ideally, one would like to detect heavy recursion automatically, and only perform apply/return inlining when needed. We conclude that, for general usage, subroutine threading plus branch inlining provides the best trade-off.

We now demonstrate that context-threaded dispatch is complementary to inlining techniques.

5.3 Inlining

Inlining techniques address the context problem by replicating bytecode bodies and removing dispatch code. This reduces both instructions executed and pipeline hazards. In this section we

show that, although both selective inlining and our context threading technique reduce pipeline hazards, context threading is slower due to the overhead of its extra dispatch instructions. We investigate this issue by comparing our own *tiny inlining* technique with selective inlining.

In Figures 5.2, 5.4 and 5.5(b) the bar labeled SABLEVM shows our measurements of Gagnon's selective inlining implementation for SableVM [31]. From these Figures, we see that selective inlining reduces both MPT and LR/CTR stalls significantly as compared to direct threading, but it is not as effective in this regard as subroutine threading alone. The larger reductions in pipeline hazards for context threading, however, do not necessarily translate into better performance over selective inlining. Figure 5.4(a) illustrates that SableVM's selective inlining beats context threading on the P4 by roughly 5%, whereas on the PPC7410 and the PPC970, both techniques have roughly the same execution time, as shown in Figure 5.4(b) and Figure 5.5(a), respectively. These results show that reducing pipeline hazards caused by dispatch is not sufficient to match the performance of selective inlining. By eliminating some dispatch code, selective inlining can do the same real work with fewer instructions than context threading.

Context threading is a dispatch technique, and can be easily combined with an inlining strategy. To investigate the impact of dispatch instruction overhead and to demonstrate that context threading is complementary to inlining, we implemented *Tiny Inlining*, a simple heuristic that inlines all bodies with a length less than four times the length of our dispatch code. This eliminates the dispatch overhead for the smallest bodies and, as calls in the CTT are replaced with comparably-sized bodies, tiny inlining ensures that the total code growth is low. In fact, the smallest inlined OCaml bodies on P4 were *smaller* than the length of a relative call instruction (five bytes). Table 5.4 summarizes the effect of tiny inlining. On the P4, we come within 1% of SableVM's selective inlining implementation. On PowerPC, we outperform SableVM by 7.8% for the PPC7410 and 4.8% for the PPC970.

Table 5.4: Detailed comparison of selective inlining (SABLEVM) vs SUB+BI+AR and TINY. Numbers are elapsed time relative to direct threading. $\triangle context$ is the the difference between selective inlining and SUB+BI+AR. $\triangle tiny$ is the difference between selective inlining and TINY (the combination of context threading and tiny inlining).

Arch	Context (SUB+BI+AR)	Selective (SABLEVM)	Tiny (T)	$\Delta context$ (SABLEVM - SUB+BI+AR)	$\Delta tiny$ (SABLEVM - TINY)
P4	0.762	0.721	0.731	-0.041	-0.010
PPC7410	0.863	0.914	0.839	0.051	0.075
PPC970	0.753	0.739	0.691	-0.014	0.048

5.4 Limitations of Context Threading

The techniques described in this chapter address dispatch and hence have greater impact as the frequency of dispatch increases relative to the real work carried out. A key design decision for any virtual machine is the specific mix of virtual instructions. A computation may be carried out by many lightweight virtual instructions or fewer heavyweight ones. Figure 5.6 shows that a Tcl interpreter typically executes an order of magnitude more cycles per dispatched virtual instruction than OCaml. Another perspective is that OCaml executes proportionately more dispatch because its work is carved up into smaller virtual instructions. In the figure we see that many OCaml benchmarks average only tens of cycles per dispatched instruction. Thus, the time OCaml spends executing a typical body is of the same order of magnitude as the branch misprediction penalty of a modern CPU. On the other hand most Tcl benchmarks execute hundreds of cycles per dispatch, many times the misprediction penalty. Thus, we expect subroutine threading to speed up Tcl much less than OCaml. Figure 5.7 reports the performance of subroutine threaded OCaml on an UltraSPARC III. As shown in the figure, subroutine threading speeds up OCaml on the UltraSPARC by about 13%. In contrast, the geometric mean of 500 Tcl benchmarks speeds up only by only 5.4%

Another issue raised by the Tcl implementation was that about 12% of the 500 program benchmark suite slowed down. Very few of these dispatched more than 10,000 virtual instructions. Most were tiny programs that executed as little as a few dozen dispatches. This

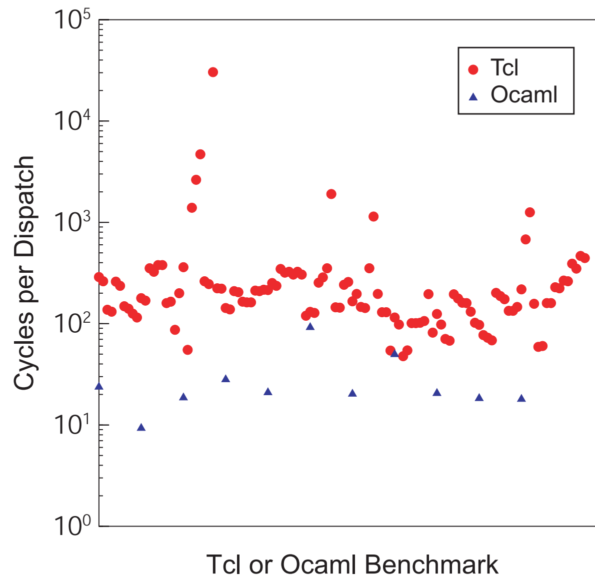


Figure 5.6: Reproduction of [76, Figure 1] showing cycles run per virtual instructions dispatched for various Tcl and OCaml benchmarks .

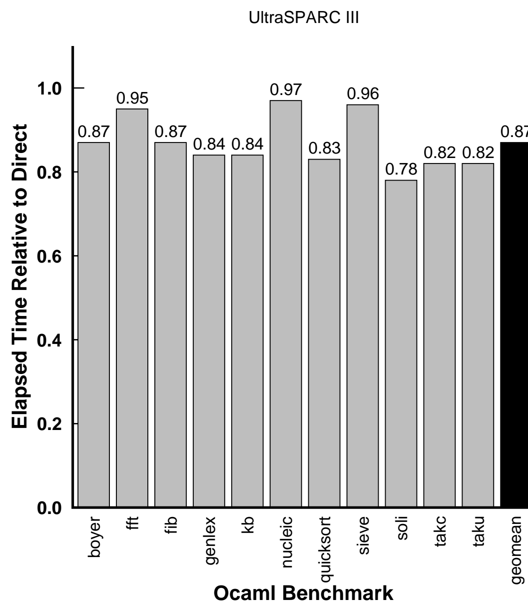


Figure 5.7: Elapsed time of subroutine threading relative to direct threading for OCaml on UltraSPARC III.

suggests that for programs that execute only a small number of virtual instructions the load time overhead of generating code in the CTT may be too high.

5.5 Chapter Summary

Our experimentation with subroutine threading has established that calling virtual instruction bodies is an efficient way of dispatching virtual instructions. Subroutine threading is particularly effective at eliminating branch mispredictions caused by the dispatch of straight-line regions of virtual instructions. Branch inlining, though labor intensive to implement, eliminates the branch mispredictions caused by most virtual branches. Once the pipelines are full, the latency of dispatch instructions becomes significant. A suitable technique for addressing this overhead is inlining, and we have shown that context threading is compatible with our "tiny" inlining heuristic. With this simple approach, context threading achieves performance roughly equivalent to, and occasionally better than, selective inlining.

These results also contain some warnings. First, our attempts to finesse the implementation of virtual branch instructions using branch replication (Section 4.3) and apply/return inlining (Section 4.4) were not successful. It was only when we resorted to the much less portable branch inlining that we improved the performance of virtual branches significantly. Second, the slowdown observed amongst a few Tcl benchmarks (which dispatched very few virtual instructions) raises the concern that even the load time overhead of subroutine threading may be too high. This suggests that we should investigate lazy approaches so we can delay generating code until it is needed.

These results inform our design of a gradually extensible interpreter, to be presented next. We suggested, in Chapter 1, that a JIT compiler would be simpler to build if its code generator has the option of falling back on calling virtual instruction bodies. The resulting fall back code is very similar to code generated at load time by a subroutine-threaded interpreter. In this chapter we have seen that linear sequences of virtual instructions program can be efficiently

dispatched using subroutine threading. This suggests that there would be little or no performance penalty, relative to interpretation, when a JIT falls back on calling sequences of virtual instructions that it chooses not to compile.

We have shown that dispatching virtual branch instructions efficiently can gain 5% or more performance. We have shown that branch inlining, though not portable, is an effective way of reducing branch mispredictions. However, our experience has been that branch inlining is time consuming to implement. In the next chapter we will show that identifying hot interprocedural paths, or traces, at runtime enables a much simpler way of dealing with virtual branches that performs as well as branch inlining.

Chapter 6

Design and Implementation of YETI

This chapter describes our gradually extensible trace interpreter, or Yeti for short. The main goal of this part of our research is to design and implement a language VM that is a simple, efficient interpreter and yet can be conveniently, and gradually, extended with a JIT compiler.

As we argued in Chapter 1, we believe the key ingredients for this are threefold. First, the system should implement callable virtual instruction bodies that can be dispatched both by the interpreter and from JIT compiler generated code. Second, the system should compile, then run, dynamically identified regions of code that contain only hot code. We pointed out that hot interprocedural paths, or traces, seem like a good choice. Third, the JIT compiler should be able to fall back on generating dispatch code to virtual instruction bodies when it encounters virtual instructions that it does not fully support. The combination of these features enables a gradual style of JIT development where compiler support for virtual instructions can be added one instruction at a time.

A similar argument can be made that the code generated for each hot region of the virtual program should also be callable and should update interpreter state before returning so that interpretation may resume immediately. We call this a *region body* because it essentially is a generated virtual instruction body for a newly created, runtime identified, virtual instruction.

Region bodies are to be called with interpreter state as the first virtual instruction in the

region would have seen it and return with the interpreter state as the last the virtual instruction would have left it. Within the region, body interpreter state need not be kept up-to-date. A region body can have multiple return points due to exceptions (in straight-line code) or trace exits.

Packaging generated code as callable also aims to support an incremental style of development, in this case allowing new and presumably larger or more highly optimized regions of the virtual program to be identified, compiled and dispatched. Currently, Yeti dispatches single virtual instruction bodies, subroutine-threaded region bodies for straight-line sections of code, and interpreted and compiled traces.

Section 6.1 gives an overview of our implementation. Section 6.2 describes how regions are identified. The runtime environment of a trace is described in Section 6.3. Section 6.4 describes how region bodies are generated for interpreted and JIT compiled traces. Finally, Section 6.5 describes ways in which of our implementation is challenged by the software environment in which it is implemented.

6.1 Structure and Overview of Yeti

Our system starts out as a simple DCT interpreter as discussed in Section 3.2. After each instruction has run once, instrumentation called from the dispatch loop identifies straight line sections of the virtual program. Simple subroutine-threaded region bodies are generated. These are installed by overwriting the DTT slot corresponding to the first virtual instruction in the region with the entry point of the new region body. Subsequently, the subroutine-threaded code executes. The system, up to this point, is operating as a lazy loaded subroutine-threaded interpreter. This alone can speed up programs with long linear blocks (like `compress` and `mpeg`) relative to direct-threaded performance.

As the program executes, profiling associates and updates event counters in a *payload* structure corresponding to each region. Eventually, hot traces are identified and translated to region

bodies. We will describe two ways traces are compiled. Interpreted traces, described in 6.4.1, implement traces in the simplest way we could conceive of, whereas JIT compiled traces, described in 6.4.2, compile the virtual instructions in each trace to register allocated native code. A novel aspect of our JIT is that compiles only a subset of virtual instructions while falling back on dispatch for the remainder. Currently, our system generates code for about 50 integer and object virtual instructions, including all of Java's conditional branch instructions. We have invested no effort in classical optimizations apart from a relatively simple variation on inlining when the invocation and return of a method occur in the same trace.

Ordinarily, DCT is slow, because it suffers a branch misprediction penalty for almost every iteration of the dispatch loop, but this turns out not to be a performance problem for Yeti. As hot region bodies are identified, installed, dispatched and linked together, execution shifts almost entirely to within the region bodies and consequently the overhead of the dispatch loop becomes negligible.

Initial Load Figure 6.1 shows how our running example (Figure 2.1) is loaded by Yeti. In the figure, the bodies are the same C coded virtual instruction bodies we show in Figure 4.2. Initially all instances of an instruction, like the two instances of `iload` in the figure, point to the same shared region bodies. This makes the initial load lightweight as no code needs to be generated and a small (static) set of region bodies and associated profiling payloads are shared by all instances of virtual instructions.

Like direct threading and regular DCT, Yeti loads each virtual instruction into one or more slots in the DTT when the virtual program is loaded. Arguments to virtual instructions are handled exactly the same as DCT or direct threading. However, we have enhanced the representation of the virtual opcode significantly. In Yeti, we add a level of indirection – the first DTT slot of each instruction points to an instance of a *dispatcher* structure instead of the address of a virtual instruction body.

Dispatcher It is the need to efficiently associate the `vPC` with *both* the body (for dispatch) and the payload (for profiling) that motivates the extra indirection in our design. The alternative would be to maintain a side table associating the payload and `vPC`. We chose the current arrangement over a hash table because it is easier to debug.

The dispatcher structure contains four key fields. The region body to be dispatched is stored in the *body* field. The *preworker* and *postworker* fields store the addresses of instrumentation functions to be called before and after the dispatch of the region body respectively. Finally, the dispatcher has a payload field, which is a region of profiling or other data that the instrumentation needs to associate with the region body. The most obvious use of the payload is to count events associated with each region body. We define specialized payload structures to describe virtual instructions, linear blocks, and traces.

When a dispatcher is created specific preworker and postworker functions are chosen depending on the type of region body the dispatcher describes. The design is object-based in the sense that the choice of a given preworker and postworker determines the behavior of the instrumentation for the given region body. In our design, the workers assume that they are always associated with a specific type of payload.

Dispatch Loop The dispatch loop, shaded in Figure 6.1, requires an extra level of indirection to call each body. The overhead of the extra indirection is of little concern as any given instruction will be executed only a few times using this generic mechanism.

Figure 6.1 also illustrates how instrumentation code for the region is called before (the *preworker*) and after (the *postworker*) the instruction body is executed. Initially instrumentation is interposed around the dispatch of each virtual instruction. This is convenient as it puts the runtime in control when the destination of each virtual branch has been determined but before it is dispatched. Later, as larger region bodies are installed, instrumentation is dispatched before and after the execution of the region body (no longer after each instruction).

An interesting feature omitted from the figure is that Yeti actually has several specialized

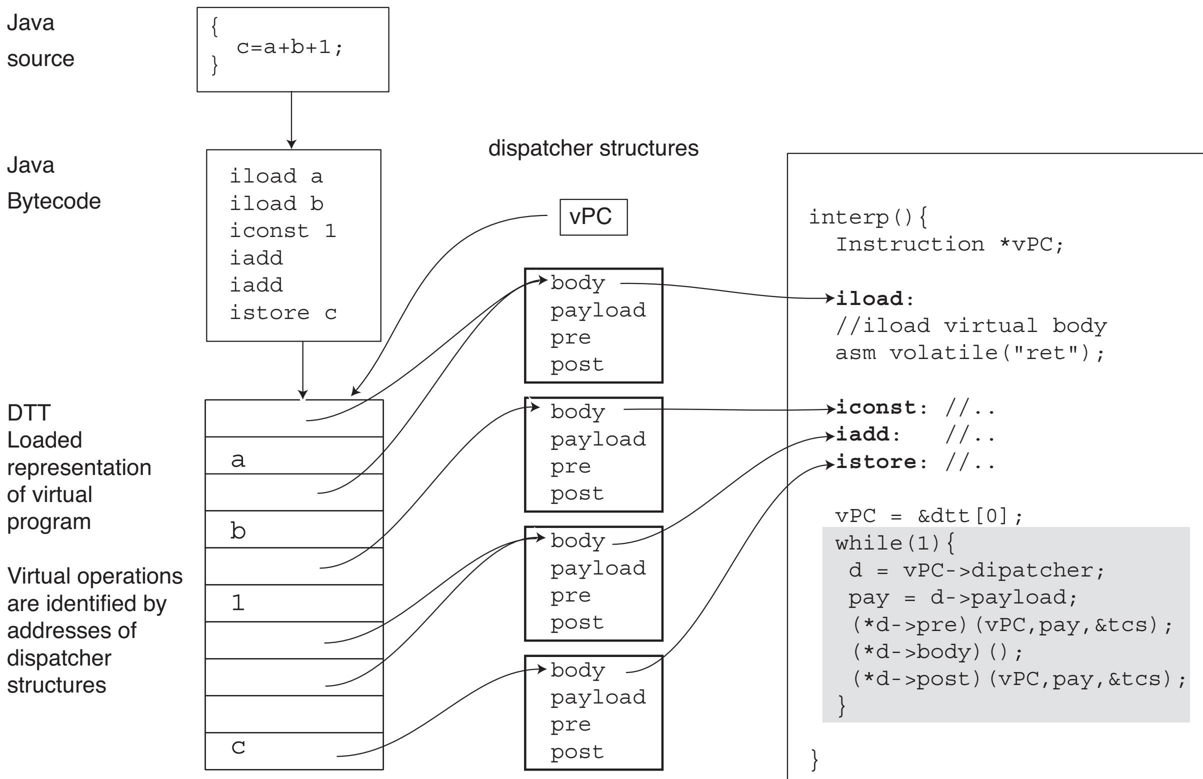


Figure 6.1: Virtual program loaded into Yeti showing how dispatcher structures are initially shared between all instances of a virtual instruction. The dispatch loop, shaded, is similar the dispatch loop of direct call threading except that another level of indirection, through the the dispatcher structure, has been added. Profiling instrumentation is called before and after the dispatch of the body.

dispatch loops. For instance, when a trace is dispatched the only remaining event to monitor is the emergence of a hot trace exit. Overhead can be significantly reduced by providing a specialized dispatch loop exclusively for traces that inlines only the required instrumentation. In general, profiling can be optimized, or turned off altogether, by changing dispatch loops.

Thread Context Structure Modern virtual machines support multiple threads of execution. Our design, like many modern interpreters, requires that each new interpreter thread runs in a separate `pthread` starting with a new invocation of the `interp` function. This means that any local variables declared in `interp` are thread-private data. The DTT, dispatchers and region bodies, on the other hand, are shared by all threads.

Yeti needs a small additional amount of thread-private data for its own purposes. To keep

all thread-private data together, we have added a new structure to the `interp` function called the *thread context structure*, or TCS. The TCS contains only a few fields, mostly in support of the region identification and trace exit profiling. For instance, in support of region identification, the TCS provides the `recordMode` bit, which indicates whether the current thread is actively recording a region; and the *history list,* that records region bodies as they are executed. Section 6.4.2 describes the role played by the TCS in profiling trace exits.

A pointer to the TCS is passed to preworker and postworkers each time they are called. For simplicity, the TCS was omitted from Figure 6.1 but appears in Figure 6.2 where it is the root of the history list.

6.2 Region Selection

Our strategy for identifying hot regions of the program is carried out by preworkers and postworkers in conjunction with state information passed in the TCS. When the profiling instrumentation discovers the beginning of a new region to be compiled into a region body it sets the `recordMode` bit in the TCS. As described below, this may be done by the preworker (as for linear blocks) or the postworker (as for traces). Once the `recordMode` bit is set the thread is actively collecting a region of the program. In this mode the preworker appends the payload of each region body about to be executed to the thread-private history list in the TCS.

Eventually a preworker or postworker will recognize that execution has reached the end of the region to be collected and clears `recordMode`. At this point a new region body is generated from the history list.

6.2.1 Initiating Region Discovery

There are two good reasons why we should ignore the first execution of each virtual instruction before considering it for inclusion in a region body. First, as discussed in Section 3.7.2, late binding languages like Java may rewrite some virtual instructions the first time they execute.

We should delay region selection until after these instructions have been rewritten. Second, some virtual instructions, for instance static class initialization blocks in Java, only execute once. This suggests that we should always wait until the second execution before considering a virtual instruction.

The obvious way of implementing this is to increment a counter the first time an instruction executes. However, this cannot be implemented with our loading strategy because a shared dispatcher has no simple way of counting how many times a specific instance has been dispatched. For example, in Figure 6.1 both instances of `iload` share the same dispatcher and payload, so there is no place to maintain a counter for each instance.

Hence, after the first execution, the preworker replaces the shared dispatcher with a new, non-shared, instance of a *block discovery dispatcher*. The second time the instruction is dispatched, the block discovery dispatcher sets about identifying *linear blocks*, as described next.

6.2.2 Linear Block Detection

A linear block is a runtime approximation of a basic block, namely a straight-line section of the virtual program ending with a branch. The process of identifying linear regions of the program is carried out by the block discovery preworker based on state information it is passed in the TCS.

We start our explanation of how the block discovery works with a detailed walk-through of how the block discovery preworker identifies a new linear block. Suppose a block discovery preworker is called for an instance of virtual instruction *i* at *vPC*. A block discovery dispatcher was installed for *i* after it executed for the first time. Hence, whenever the block discovery preworker is called there are two possibilities. If `recordMode` is set then *i* should simply be appended to the history list (in the TCS) and thus added to the linear region currently being recorded[1]. Otherwise, if `recordMode` is clear, then *i* must begin a new linear block. (If there already was a linear region starting at *vPC* then a dispatcher for that region body would have

[1]There are corner cases, for instance, if *i* is encountered while a trace is being collected.

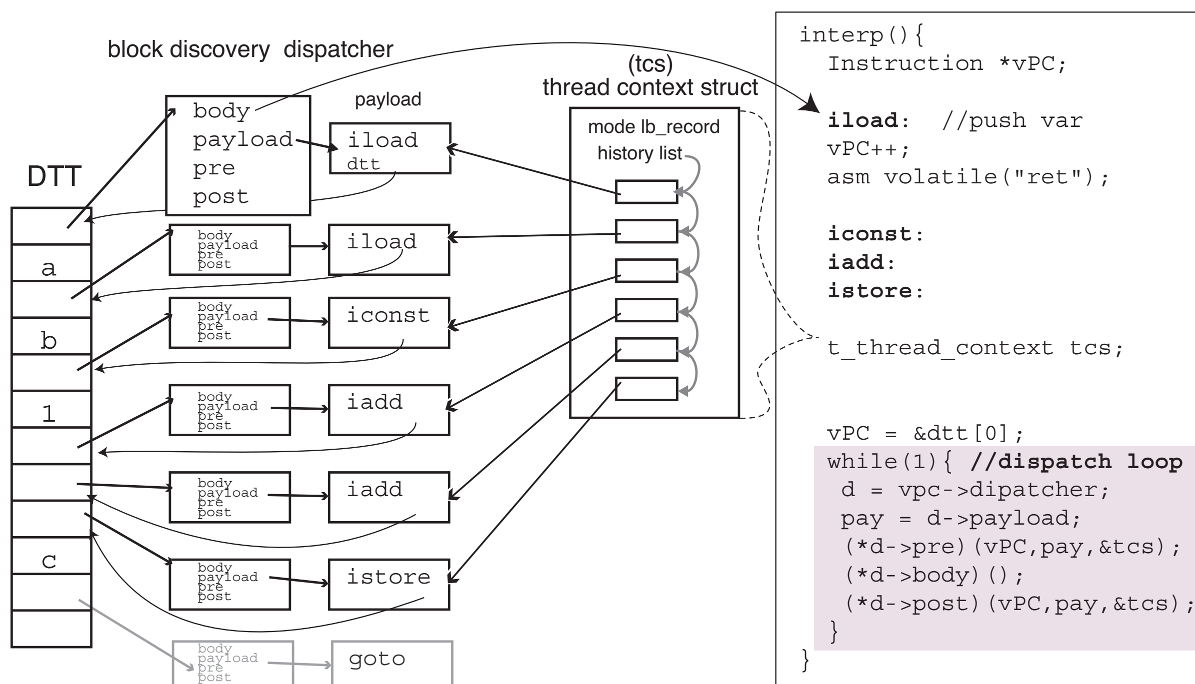


Figure 6.2: Shows a region of the DTT during block recording mode. The body of each block discovery dispatcher points to the corresponding virtual instruction body (Only the body for the first iload is shown). The dispatcher's payload field points to instances of instruction payload. The thread context struct is shown as tcs.

executed instead.)

The preworker recognizes the end of the linear region when it encounters a virtual branch instruction. At this point `recordMode` is cleared and a new subroutine-threaded region body is generated from the instructions on the history list. Figure 6.2 illustrates an intermediate stage during the identification of the linear block of our running example. The preworker has appended the payload of each instruction onto the thread's history list, rooted in the TCS. In the figure, a branch instruction, a `goto,` will end the current linear block.

Figure 6.3 illustrates the situation just after the collection of the linear block. The dispatcher corresponding to the entry point of the linear block has been replaced by a new *linear block dispatcher* whose job it will be to search for traces. The linear block dispatcher points to a new payload created from the history list; its body field points to a subroutine-threading-style region body that has been generated for the linear block. Note that linear blocks are not basic blocks because they do not end at labels. If the virtual program later branches to a virtual

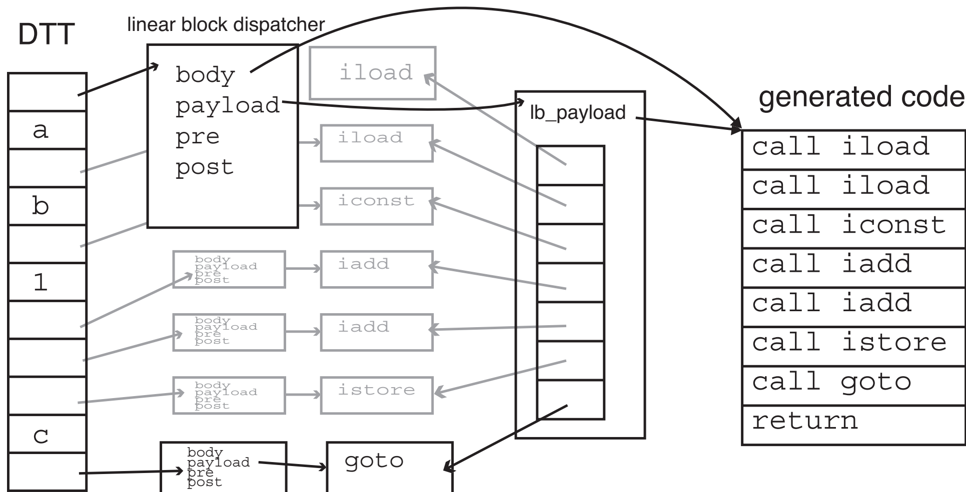


Figure 6.3: Shows a region of the DTT just after block recording mode has finished.

address that happens to be in the middle of a linear block our system will create a new linear block that replicates the tail of the original.

6.2.3 Trace Selection

The postworker of a linear block dispatcher is called after the last virtual instruction of the linear block has executed. Since, by definition, linear blocks end with branches, after executing the last instruction the `vPC` has been set to the destination of the branch and hence points to one of the successors of the linear block. The postworker runs at exactly the right moment to profile edges of the control flow graph, namely after each branch destination is known, and yet before the destination is executed.

If the `vPC` of the destination is *less* than the `vPC` of the virtual branch instruction itself, this is a reverse branch – a likely candidate for the latch of a loop. According to the heuristics developed by Dynamo (see Section 2.5), hot reverse branches are good places to start the search for hot code. Accordingly, when our system detects a reverse branch that has executed 100 times[2] it enters *trace recording mode*. In trace recording mode, similar to linear block

[2]Performance does not seem sensitive to the particular value, so we chose a round number in the vicinity of the value used by Dynamo.

recording mode, the postworker adds each linear block payload to the thread's history list. The situation is very similar to that illustrated in Figure 6.2, except the history list describes linear blocks instead of virtual instructions. Our system, like Dynamo, ends a trace (i) when it reaches a reverse branch or finds a cycle, or (ii) when it contains too many (currently 100) linear blocks.

When trace generation ends, a new *trace dispatcher* is created and installed. This is quite similar to Figure 6.3 apart from the need to support trace exits. The payload of a trace dispatcher includes a table of *trace exit descriptors*, one for each linear block in the trace. See Figure 6.4.

Although code could be generated for the trace at this point, we postpone code generation until the trace has run a few times, currently five, in trace training mode[3]. Trace training mode uses a specialized dispatch loop that calls additional instrumentation before and after dispatching each virtual instruction in the trace. The instrumentation is passed pointers to various interpreter variables (top of the expression stack, a description of the currently executing method, etc). In principle, almost any detail of the virtual machine's state can be recorded. Currently, we record the class of every Java object upon which a virtual method is invoked.

Once the trace has been trained we generate and install a region body. We have implemented two different mechanisms for generating code for a trace. Early in the project we implemented a simple approach, *interpreted traces*, that generates very simple subroutine-threaded style code for each trace. Then, with a great deal more effort, we implemented our trace-based JIT compiler. Both approaches are described in Section 6.4.

Before we discuss code generation, we need to describe the runtime of the trace system and especially the operation of trace exits.

[3]As almost all the callsites in the SPECjvm98 benchmarks are monomorphic, a smaller number of training runs would have been sufficient but unrealistic.

6.3 Trace Exit Runtime

One of the properties that make traces a desirable shape of region body is that they predict hot paths through the virtual program. If the predictions are good, and the Dynamo results suggest that they are, we assume that most trace exits are not taken. The trace exits that are taken, however, quickly become hot and hence new traces must be generated and linked. This means that it will likely pay to burden the implementation of a trace exit with some extra overhead if this makes the path through the trace more efficient.

We use a combination of code generation (in the region body for the trace) and runtime profiling instrumentation (in the postworker called after each trace returns to the dispatch loop) to detect which trace exits are occurring and what to do about it.

Trace exits occur when execution diverges from the path collected during trace generation, in other words, when the destination of a virtual branch instruction in the trace is different from what was recorded during trace generation. Generated trace exit code in the trace detects the divergence and branches to a *trace exit handler*. Generated code in the trace exit handler records which trace exit has occurred by storing, into the TCS, the address of the trace payload (to identify the trace) and the index of the trace exit (to identify the specific branch). The trace exit handler then returns to the dispatch loop, which, as usual, calls the postworker. The postworker uses the information in the TCS to update the trace exit profiling information in the trace payload.

This scheme minimizes overhead for traces that complete or link at the expense of cold trace exits. Conceptually, the postworker has only a few alternative to chose from:

1. If the trace exit is still cold, increment the counter corresponding to the trace exit in the trace payload.

2. Notice that the counter has crossed the hot threshold and arrange to generate a new trace.

3. Notice that a trace already exists at the destination and link the trace exit handler to the destination trace.

Alternative 1 is trivial, the postworker increments a counter and returns. Alternative 2 is also simple, the postworker simply sets the `recordMode` bit in TCS and the destination trace will start being collected immediately. Alternative 3 is more challenging and will be described in the next section.

6.3.1 Trace Linking

The goal of trace linking is to rewrite the trace exit handler of a hot trace exit to branch directly to the destination trace rather than return to the dispatch loop. The actual mechanism we use depends on the underlying virtual branch instruction. There are two main cases, branches with only one off-trace destination and branches with multiple off-trace destinations.

Regular conditional branches, like Java's `if_icmp`, are quite simple. The branch has only two destinations, one on the trace and the other off. When the trace exit becomes hot a new trace is generated starting with the off-trace destination. Then, the next time the trace exit occurs, the postworker links the trace exit handler to the new trace by rewriting the branch instruction in the trace exit handler to jump directly to the destination trace instead of returning to the dispatch loop. Subsequently, execution stays in the code cache for both paths of the program.

Multiple destination branches, like method invocation and return, are more complex. When a trace exit originating from a multi-way branch occurs we are faced with two additional challenges. First, profiling multiple destinations is more expensive than just maintaining one counter. Second, when one or more of the possible destinations are also traces, the trace exit handler needs some mechanism to jump to the right one.

The first challenge we essentially ignore. We use a simple counter and trace generate *all* destinations of a hot trace exit that arise. The danger of this strategy is that we could trace generate superfluous cold destinations and waste trace generation time and code cache memory.

The second challenge concerns the efficient selection of a destination trace to which to link, and the mechanism used to branch there. To choose a destination, we follow the heuristic

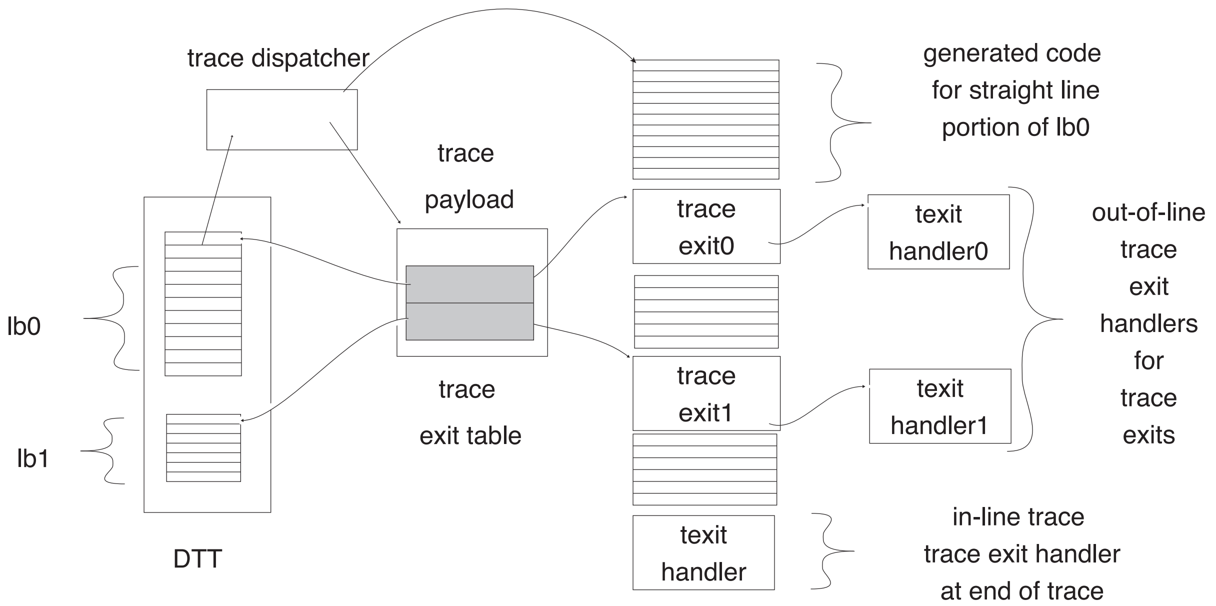


Figure 6.4: Schematic of a trace illustrating how trace exit table (shaded) in trace payload has recorded the on-trace destination of each virtual branch

developed by Dynamo for regular branches – that is, we link to destinations in the order they are encountered. The rational is that the highest probability trace exits will occur sooner. At link time, we rewrite the code in the trace exit handler with code that checks the value of the `vPC`. If it equals the `vPC` of a linked trace, we branch directly to that trace; otherwise we return to the dispatch loop. Because the specific values of the `vPC` for each destination trace are visible to the postworker, we can hard-wire the comparand in the generated code. In fact, we can generate a sequence of compares checking for each of the multiple destinations in turn. Eventually, a sufficiently long cascade would perform no better than a trip around the dispatch loop. Currently we limit ourselves to two linked destinations per trace exit. This mechanism is similar to the technique used for interpreted traces, described next.

6.4 Generating code for traces

Generating code for a trace is made up of two main tasks, generating the main body of the trace and generating a trace exit handler for each trace exit. After trace selection the TCS history list

contains a list of linear block payloads that were selected. By traversing the list we can visit each virtual instruction in the trace.

We describe two different strategies for compiling a trace. Both schemes use the same runtime and carry out trace linking identically. Interpreted traces, described next, represent our simplest approach to generating code for a trace. JIT compiled traces, described in Section 6.4.2, contain a mixture of compiled code and dispatch.

Figure 6.4 gives a schematic for a hypothetical trace. As shown in the figure, the dispatcher is the root of the data structure and points to the payload and the entry point of the region body. The payload contains a counter (not shown in the figure) and a trace exit table. The trace exit table is an array of trace exit descriptors, one for each trace exit in the trace. Each trace exit descriptor contains a counter (not shown) and a pointer to the trace exit handler for each trace exit. The counter is used to determine when a trace exit becomes hot. The pointer to the trace exit handler is used to mark the location that will be rewritten for trace linking.

6.4.1 Interpreted Traces

Interpreted traces require only slightly more complex code generation than subroutine threading, but are about as effective as branch inlining (See Section 4.3) at reducing the overhead of dispatching virtual branch instructions. We call them interpreted because no virtual instruction bodies are compiled in-line, rather, an interpreted trace dispatches all virtual instruction bodies including virtual branches.

The trace payload identifies each linear block in the trace and each linear block payload lists every virtual instruction. Hence, by iterating over the linear block payloads the straight line portions of a trace can be easily implemented as regions of subroutine-threaded code.

Trace exits require only slightly more complicated code generation. A trace is a hot path through the virtual program, or put another way, a trace predicts the value of the `vPC` after each of its constituent virtual branch instructions has executed. Taking this view, the purpose of each trace exit is to ensure that the branch it guards has set the `vPC` to the on-trace destination.

The on-trace destination of each virtual branch is recorded in the trace payload as the trace is generated. Hence, the simplest possible implementation of a trace exit must do three things. First, it dispatches the virtual branch body. Second, it compares the value of the `vPC,` the destination of the branch, to the on-trace `vPC` predicted by the trace. A compare immediate can be used, since the on-trace value of the `vPC` is known and is constant. Third, it conditionally branches to the trace exit handler if the comparison fails.

This code is somewhat reminiscent of the branch replication technique we described in Section 4.3 except that instead of following the dispatch of the virtual branch body with an expensive *indirect* branch we generate a compare immediate followed by a *direct conditional* branch to the trace exit handler. We expect this technique to be quite easy for the branch predictors of the underlying processor to predict because the direct conditional branch is fully exposed to the branch history predictors. As we shall show in the next chapter, interpreted traces achieve a level of performance similar to subroutine threading plus branch inlining.

6.4.2 JIT Compiled Traces

Our JIT does not perform any classical optimizations and does not build any internal representation before compiling a trace. As traces contain no merge points, we perform a single pass through each trace allocating expression stack slots to registers and generating code.

An important aspect of our JIT design is that it can generate code for a trace before it supports all virtual instructions. Our JIT generates register allocated machine code for contiguous sequences of virtual instructions it recognizes. When an unfamiliar virtual instruction is encountered, code is generated to flush any temporary values held in registers back to the Java expression stack. Then, the bodies of any uncompilable or unfamiliar virtual instructions are dispatched using subroutine threading. This significantly eases development as the compiler can be extended one virtual instruction at a time. The same tactics can be used for virtual instructions that the JIT partially supports. When the compiler encounters an awkward corner case it can simply give up and fall back to subroutine dispatch instead.

Expression stack slots are assigned to registers, freeing the generated code from maintaining the expression stack. Immediate arguments to virtual instructions, normally loaded from the DTT, are loaded into registers using load immediate instructions whenever possible. This frees the generated code from the maintaining the `vPC`.

Machine code generation is performed using the `ccg` [57] runtime assembler.

Dedicated Registers

The code generated by Yeti must be able to load and store values to the same Java expression stack and local variable array referred to by the C code implementing the virtual instruction bodies. Our current PowerPC implementation side-steps this difficulty by dedicating hardware registers for the values that must be shared between our generated code and C generated bodies. At present we dedicate registers for the `vPC`, the top of the Java expression stack and the pointer to the base of the local variables. Code is generated to adjust the value of the dedicated registers as part of the flush sequence, described below.

On targets with fewer registers, notably Intel's Pentium, there may not be enough general purpose registers to dedicate three of them for our own purposes. There, we generate code that accesses the variables in memory.

Register Allocation

Java virtual instructions, and those of many other virtual machines, pop arguments off and push results onto an expression stack (See Section 2.1.1). Naive compilation of the pushes and pops would result in many redundant loads, stores and adjustments of the pointer to the top of the expression stack. Our JIT assigns the temporary values to registers instead.

Our register allocator and code generator are combined and perform only one pass. As we examine each virtual instruction we maintain a compile time structure we call the *shadow stack*. The shadow stack associates each value in an expression stack slot with the register to which it has been assigned. Whenever a virtual instruction would pop one of its inputs we first

check if there already is a register for that value in the corresponding shadow stack slot. If so, we use the register instead of generating any code to pop the expression stack. Similarly, whenever a virtual instruction would push a new value onto the expression stack we assign a new register to the value and push this on the shadow. We forgo generating any code to push the value onto the expression stack.

A convenient property of this approach is that every value assigned to a register always has a *home location* on the expression stack. If we run out of registers we simply spill the register whose home location is deepest on the shadow stack (as all the shallower values will be needed sooner [58]).

Flushing Registers to Expression Stack

The simple strategy for assigning expression stack slots to registers we have described assumes that execution remains on the trace and that all instructions have been compiled. However, when a trace exit is taken, or when the JIT needs to fall back to calling a virtual instruction body, all values in registers must be saved back to the expression stack.

Flush code is generated by scanning the shadow stack to find every expression stack slot currently assigned to a register. A store is generated to store each such live register to its home location on the expression stack. Then, the shadow stack is reinitialized to empty and all registers are marked as free.

Generated code typically does not need to maintain the dedicated registers, for instance the top of the expression stack, or the `vPC`, until it is about to return to the interpreter. Generated flush code updates the values held by the dedicated registers as well.

Trace Exits and Trace Exit Handlers

The virtual branch instruction ending each block is compiled into a trace exit. We follow two different strategies for trace exits. The first case, regular conditional branch virtual instructions, are compiled by our JIT into machine code that conditionally branches to a trace exit handler

when execution would leave the trace. The generated code implements the semantics of the virtual instruction body, and compares and conditionally branches on the values in registers. It does not access the vPC. PowerPC code for this case appears in Figure 6.5. The sense of the conditional branch is adjusted so that the branch is always not-taken for the on-trace path. The second case, for more complex virtual branch instructions, such as for method invocation and return, which may have multiple destinations, are handled as for interpreted traces. (Polymorphic method dispatch is also handled this way if it cannot be optimized as described in Section 6.4.3.)

Trace exit handlers have two further roles. First, since compiled traces contain compiled code, it may be necessary to flush values held in registers and update the values of dedicated registers. For instance, in Figure 6.5, the trace exit handler adjusts the vPC. Flush code is the only difference between trace exit handlers for interpreted and compiled traces. Second, trace linking is achieved by overwriting code in a trace exit handler. (This is the only situation in which we rewrite code.) To link traces, the tail of the trace exit handler is rewritten to branch to the destination trace rather than return to the dispatch loop.

The trace link branch occurs after the flush code which means that registers are flushed only to be reloaded by the destination trace. We have not yet implemented any optimization to address this redundancy. However, if the shadow stack at the trace exit were to be saved aside it could be used to prime the compilation of the destination. Then, the trace link could be inserted before the flush code.

Most trace exit handlers are reached only when a conditional trace exit is taken. The only exception occurs when a trace executes to completion. Then, control must return to the dispatch loop. To implement this each trace ends with an in-line trace exit handler. Like any other trace exit handler, it may later be linked to its destination trace if one becomes hot.

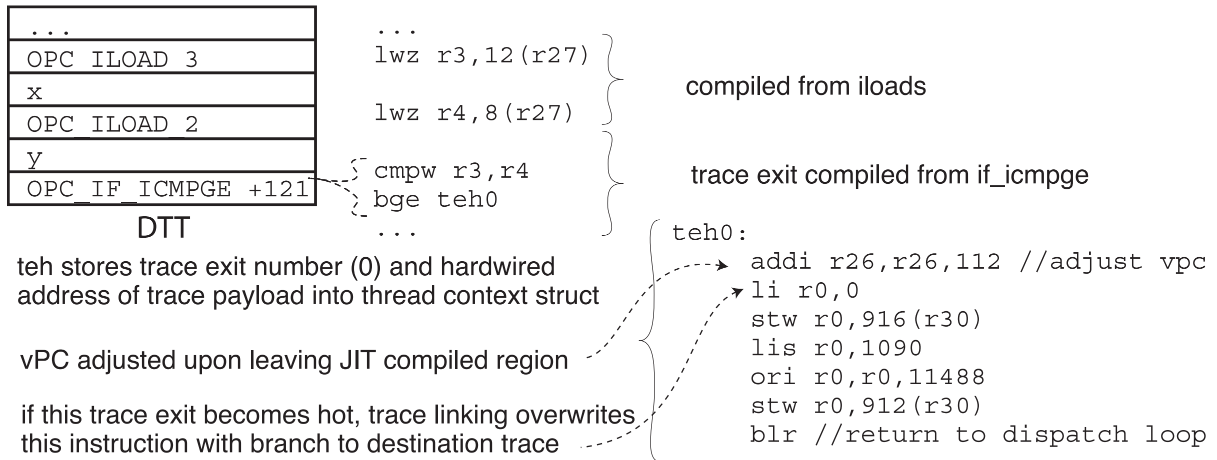


Figure 6.5: PowerPC code for a portion of a trace region body, showing details of a trace exit and trace exit handler. This code assumes that r26 has been dedicated for the `vPC`. In addition the generated code in the trace exit handler uses `r30`, the stack pointer as defined by the ABI, to store the trace exit id into the TCS.

6.4.3 Trace Optimization

We describe two optimizations here: how loops are handled and how the training data can be used to optimize method invocation.

Inner Loops

An intrinsic property of Dynamo's trace selection heuristic is that the innermost loops of a program are often selected into a single trace ending with the loop closing reverse branch. This occurs because trace generation starts at the target of reverse branches and ends whenever it reaches a reverse branch. Note that there may be many branches, including calls and returns, along the way. When the trace is compiled the loop is trivial to find because the last virtual instruction in the trace is a virtual conditional branch back to its entry.

Inner loops expose a problem with the way we end a trace. Normally, a trace exit is compiled as a branch taken to the trace exit handler for the off-trace path and a fall-through for the on-trace path. If this approach were followed, each iteration of a hot inner loop would execute to the inline trace exit handler at the end of the trace and return to the dispatch loop. Soon this trace exit would become hot and trace linking would rewrite the inline trace exit to branch

back to the head of the trace. To avoid the extra branch and pointless trace linking the trace JIT compiles a reverse branch differently – reversing the sense of the trace exit and generating a reverse conditional branch back to entry point of the trace.

Thus far, we have not exploited this information to optimize the body of the trace. For example, it would be relatively easy to detect loop invariant instructions and move them to a newly constructed loop preheader. However, the flow graph of the resulting unit of compilation would then include a merge point because the head of the loop would have two inbound edges (the back edge and the edge from the preheader). The register allocation scheme we have described does not support merge points.

Virtual Method Invocation

So far, all the trace exits we have described have been translations of virtual branch instructions. However, a trace exit can be used to guard other speculative optimizations as well. Our strategy for optimizing virtual method invocation is to generate a guard trace exit that is much cheaper than a full method dispatch. If the guard code falls through, we know execution should continue along the trace.

Specifically, if the class of the invoked-upon object is different than recorded when the trace was generated, a trace exit must occur. At trace generation time we know the on-trace destination of each call. From the training profile we know the class of each invoked-upon object. Thus, we can easily generate a *virtual invoke guard* that branches to the trace exit handler if the class of the object on top of the expression stack is not the same as recorded during training. Then, we can generate code to perform a faster, stripped down version of method invocation. The savings are primarily the work associated with looking up the destination given the class of the receiver. This technique was independently invented by Gal et al [32].

Inlining

Traces are agnostic towards method invocation and return, treating them like any other multiple-destination virtual branch instructions. However, when a return corresponds to an invoke in the same trace, the trace compiler can sometimes remove almost all method invocation overhead. Consider when the code between a method invocation and the matching return is relatively simple, for instance, it does not touch the callee's stack frame (other than the expression stack), it cannot throw an exception and it makes no further method invocations. Then, we can eliminate the invoke altogether and the only method invocation overhead that remains is the virtual invoke guard. If the inlined method body contains any trace exits the situation is slightly more complex. In this case, in order to prepare for a return somewhere off-trace, the trace exit handlers for the trace exits in the inlined code must modify the expression stack exactly as the (optimized away) method invocation would have done.

6.5 Other implementation details

Our system, as described in this chapter, generates code that coexists with virtual instruction bodies written in C. Consequently, the generated code must be able to access a few interpreter variables like the `vPC`, the top of the expression stack, and the base of the local variable array. For these heavily used interpreter variables, on machines with sufficient general purpose registers, we take the obvious approach of assigning the variables to dedicated registers. Dedicating the register might even improve the quality of code generated by the compiler for the interpreter. We note that on the PowerPC OCaml dedicates registers for the `vPC` and a few other commonly used values, presumably because it performs better this way.

A related challenge arises in our implementation of trace exit handlers. We want on-trace execution to be free of trace exit related overhead. At the same time, we need a way of recording which trace exit has occurred so that we can determine which trace exits are hot. This means that each trace exit handler, which is a region of code specific to a trace exit generated

by Yeti, must have a way of writing into the TCS. On the PowerPC we could dedicate yet another register to point to the TCS. However, this could only hurt the performance of the virtual instruction bodies, since they never refer to the TCS. Instead, we indulge in some unwarranted chumminess with gcc. Using a trick invented by Ben Vitale, we use gcc inline `asm` statements to obtain a string containing assembler gcc would generate to access the desired field in the TCS [76]. Then, we parse the string and extract all the information we need to generate code to access the field.

Our use of a dispatch loop similar to Figure 6.2, in conjunction with making virtual bodies callable by inserting inlined assembler return instructions, results in a control flow graph that is not apparent to the optimizer. First, the optimizer cannot know that the label at the head of each virtual instruction body can be reached by the function pointer call in the dispatch loop. (The compiler assumes, quite reasonably, that the function pointer call only reaches the entry point of functions.) Second, the optimizer does not know that control flows from the inlined return instruction back to the dispatch loop. We work around these difficulties by inserting computed goto's (which never actually execute) to simulate the missing edges.

6.6 Chapter Summary

In this chapter we have described the design trajectory for a high-level language virtual machine that extends from a very simple interpreter through a high-performance trace-based interpreter, to a extensible trace-based JIT compiled system. Our design goals are much more ambitious than in the preceding two chapters. There, we concentrated on how an interpreter can be made more efficient. In this chapter we presented a design that supports the evolution of a high-level language VM from a simple interpreter to a JIT. Thus, we favour infrastructure that supports the development of a JIT, for instance our dispatcher-based instrumentation, over infrastructure that merely speeds up interpretation.

An aspect of context threading that is somewhat unpalatable is that the effort invested im-

plementing branch inlining, apply/return inlining and tiny inlining does nothing to facilitate the later addition of a JIT compiler. For instance, implementing branch inlining in the interpreter runs the risk of being a throw-away effort – if evolving performance requirements eventually lead to the implementation of a JIT, then a good deal of the the effort spent building branch inlining will have to be duplicated.

In contrast to this, Yeti builds its advanced interpretation techniques on top of infrastructure that is intended to facilitate the addition of a JIT. For instance, interpreted traces require trace-based profiling that is also required to support the trace-based JIT. As we will show in the next chapter interpreted traces perform just as well as branch inlining.

With the resources at our disposal it is not feasible to show that the performance potential of our trace-based JIT compiler is equal to an optimizing method-based JIT like those deployed by Sun or IBM. Our design is intended to support any shape of region body, so in a sense the peak performance of traces is not a limiting factor, since with sufficient engineering effort peak performance could always be achieved by compiling inlined method nests.

Instead, we concentrated our JIT compiler design efforts on how to support only a subset of virtual instructions, added one at a time. We found this was a convenient way to work, much easier than bringing up a regular compiler, since interactions between code generation bugs were much reduced. Currently our JIT consists of only about 2000 statements of C source code, about half machine dependent, and compiles about 50 integer virtual instructions. Nevertheless, as we will show in the next chapter, our JIT improves the performance of the SPECjvm98 benchmarks by about 24% over interpreted traces.

The main problem with the implementation of our prototype is that our generated code depends too heavily on gcc. There are two main issues. First, our generated code occasionally needs to access interpreter values. On the PowerPC we were able to side-step the potential difficulties by dedicating registers for key interpreter variables, but clearly another approach will be necessary for 32 bit Intel processors, which have too few general purpose registers to dedicate any to interpreter variables. Second, the way we have packaged virtual instruction

bodies, and called them via a function pointer, (Figure 6.1) hides the true control flow of the interpreter from the C optimizer. We will discuss how this might be avoided by packaging bodies as nested functions in Chapter 8.

Next, in Chapter 7, we will evaluate the performance of our prototype.

Chapter 7

Evaluation of Yeti

In this chapter we evaluate Yeti from three main perspectives. First, we evaluate the effectiveness of traces for capturing the execution of regions of Java programs, and verify that the frequency of dispatching region bodies does not burden overall performance. Second, we confirm that the performance of the simplest, entry level, version of our system is reasonable, and that performance improves as more sophisticated shapes of region bodies are identified and effort is invested in compiling them. The goal here is to determine whether the first few stages of our extensible system are viable deployment candidates for an incrementally evolving system. Third, we attempt to measure the extent to which our technique is affected by various pipeline hazards, especially branch mispredictions and instruction cache misses.

We prototyped Yeti in a Java VM (rather than a language that does not have a JIT) in order to compare our techniques against high-quality implementations on well-known benchmarks. We show that through four stages of extending our system, from a simple direct call-threaded (DCT) interpreter to a trace based JIT compiler, performance improves steadily. Moreover, at each stage, the performance of our system is comparable to other Java implementations based on different, more specific techniques. Thus, DCT, the entry level of Yeti, is roughly comparable to switch threading. Interpreted traces are faster that direct threading and our trace based JIT is 27 % faster than selective inlining in SableVM.

These results indicate that our design for Yeti is a good starting point for an extensible infrastructure whose performance can be incrementally improved, in contrast to techniques like those described in Chapters 3 and 4 which are end points with little infrastructure to support the next step up in performance.

Section 7.1 describes the experimental set-up. We report the extent to which different shapes of region enable execution to stay within the code cache in Section 7.2. Section 7.3 reports how the performance of Yeti is effected by different region shapes. Section 7.4 describes preliminary performance results on the Pentium. Finally, Section 7.5 studies the effect of various pipeline hazards on performance.

7.1 Experimental Set-up

The experiments described in this section are simpler than those described in Chapter 5 because we have modified only one Java virtual machine, JamVM. Almost all our performance measurements are made on the same PowerPC machine, except for a preliminary look at interpreted traces on Pentium.

We took a different tack to investigating the micro-architectural impact of our techniques than the approach presented in Chapter 5. There, we measured specific performance monitoring counters, for instance, the number of mispredicted taken branches that occurred during the execution of a benchmark. Here, we evaluate Yeti's impact on the pipelines using a much more sophisticated infrastructure, GPUL, which determines the causes of various stall cycles.

todo:GPUL cite

Virtual Machines Yeti is a modified version of Robert Lougher's JamVM 1.1.3, which is a very neatly written Java Virtual Machine [52]. On all platforms (OSX 10.4, PowerPC and Pentium Linux) we built both our modifications to JamVM and JamVM as distributed using gcc 4.0.1.

todo: some measure of code complexity would be nice

We compare the performance of Yeti to several other JVM configurations:

Table 7.1: SPECjvm98 benchmarks including elapsed time for baseline JamVM (i.e., without any of our modifications), Yeti and Sun HotSpot.

Benchmark	Description	Elapsed Time (sec)		
		JamVM	Yeti	HotSpot
		1.3.3 direct threaded	trace JIT	1.05_6_64 optimizing JIT
compress	Lempel-Ziv	98	44	8.0
db	Database functions	56	35	23
jack	Parser generator	22	14	5.4
javac	JDK 1.0.2	33	24	9.9
jess	Expert Shell System	29	19	4.4
mpeg	read MPEG-3	87	36	4.6
mtrt	Two thread raytracer	30	25	2.1
raytrace	raytracer renderer	29	17	2.3
scimark	FFT, SOR,LU, 'large'	145	58	16

Table 7.2: Guide to labels which appear on figures and references to technique descriptions.

Technique	Label on Figures	Section describing Technique
Subroutine Threading	SUB	Section 4.2
Direct Call Threading	DCT	Section 6.1
Linear Blocks	LB	Section 6.2.2
Interpreted Traces	i-TR	Section 6.4.1
Interpreted Traces with linking OFF	i-TR-nolink	as above
Yeti - Trace JIT	TR-JIT	Section 6.4.2
SableVM 1.1.8	SABLEVM	Section 3.7.2

1. JamVM configured for direct threading (its default configuration) is our baseline because direct threading is a commonly deployed high performance dispatch technique.

2. JamVM configured to be switch threaded as an example of an entry-level interpretation technique. Many production language virtual machines have been usefully deployed using switch threading.

3. A subroutine threaded version of JamVM.

4. SableVM with selective inlining as an example of an advanced interpreter technique.

5. Sun's Hotspot JVM version 1.05 as a state of the art Java JIT.

Elapsed Time Data Elapsed time performance data was collected on a dual CPU 2 GHz PowerPC 970 processor with 512 MB of memory running Apple OSX 10.4. Pentium performance was measured on a Intel Core 2 Duo E6600 2.40GHz 4M with 2GB of memory under Linux 2.6.9. Performance is reported as the average of three measurements of elapsed time, as printed by the `time` command.

Benchmarks Table 7.1 briefly describes each SPECjvm98 benchmark [65] and `scimark`, a scientific program. Since the rest of the figures in this chapter will report performance relative to unmodified JamVM 1.1.3, Table 7.1 includes, for each benchmark, the raw elapsed time for JamVM, Yeti (running our JIT), and version 1.05.0_6_64 of Sun Microsystems' Java HotSpot JIT. (We provide the elapsed time here because below we will report performance relative to direct threaded JamVM.)

Table 7.2 provides a key to the acronyms used as labels in the following graphs and indicates the section of this thesis each technique is discussed.

Pipeline Hazards In Section 7.5 we describe how Yeti is effected by common processor pipeline hazards such as branch mispredictions and instruction cache misses. We use a new infrastructure called GPUL, built and operated by our colleagues at the Electrical Engineering Computer Group, that heuristically attributes stall cycles to various causes. We collected the GPUL data on a slightly different model of PowerPC, a 2.3 GHz PowerPC 970FX (Apple G5 Xserve) running Linux version 2.6.18. The 970FX part is a 90nm implementation of the 130nm 970, more power efficient but identical architecturally. The platform change was forced upon us because GPUL requires both Linux and a system running the new FX version of the processor.

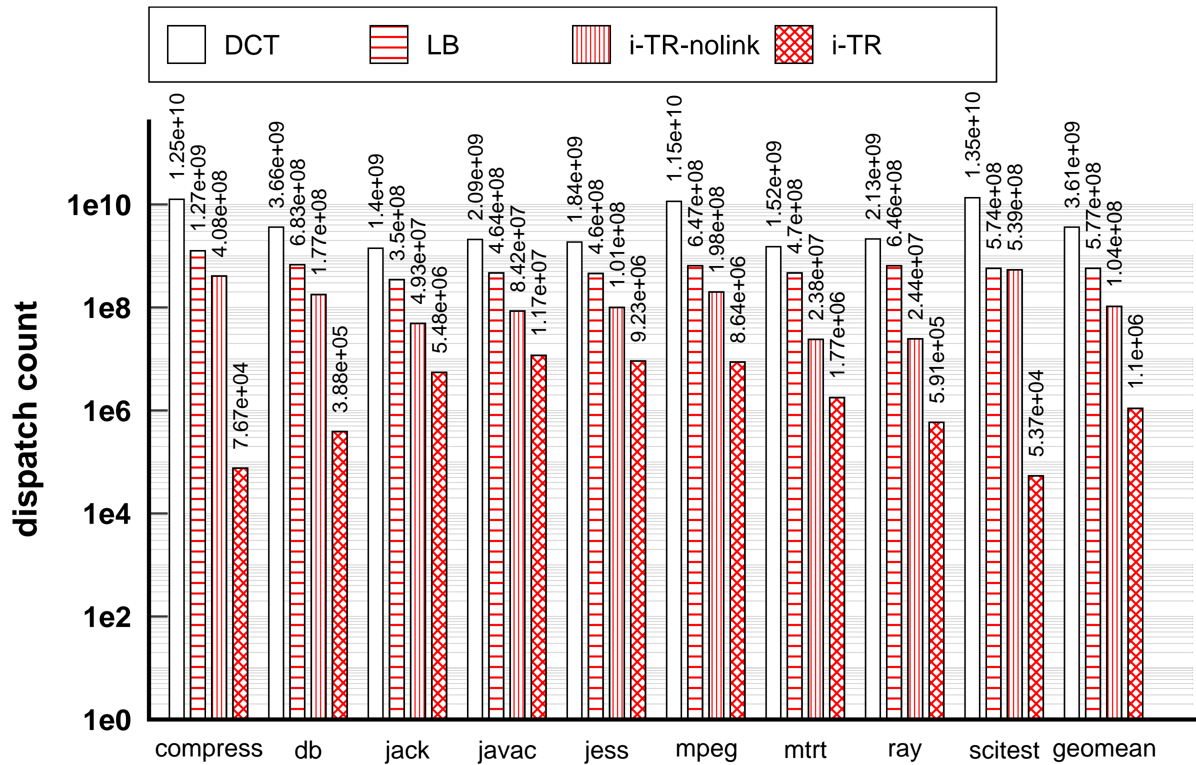


Figure 7.1: Number of dispatches executed vs region shape. The y-axis has a logarithmic scale. Numbers above bars, in scientific notation, give the number of regions dispatched. The X axis lists the SPECjvm98 benchmarks in alphabetical order.

7.2 Effect of region shape on dispatch

In this section we report data obtained by modifying Yeti's instrumentation to keep track of how many virtual instructions are executed from each region body and how often region bodies are dispatched. These data will help understand to what extent execution remains in the code cache for differently shaped regions of the program.

For a JIT to be effective, execution must spend most of its time in compiled code. We can easily count how many virtual instructions are executed from interpreted traces and so we can calculate what proportion of all virtual instructions executed come from traces. For `jack`, traces account for 99.3% of virtual instructions executed. For all the remaining benchmarks, traces account for 99.9% or more.

A remaining concern is how often execution enters and leaves the code cache. In our

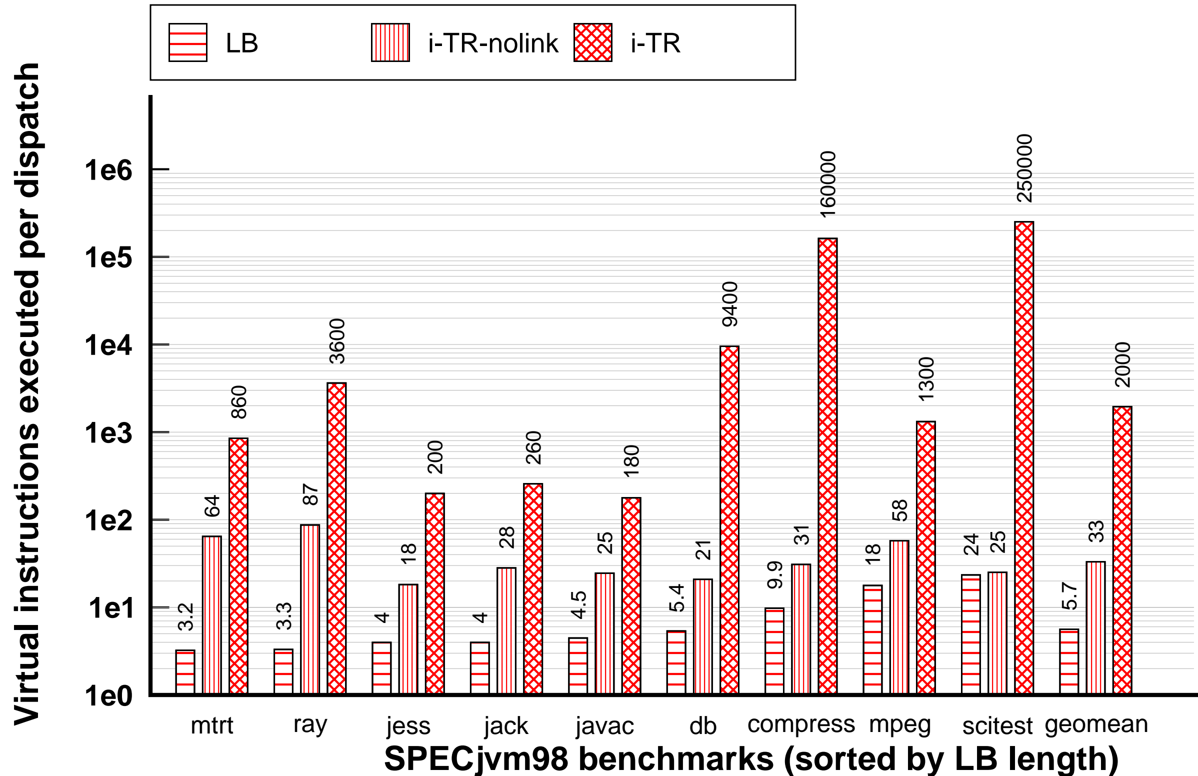


Figure 7.2: Number of virtual instructions executed per dispatch for each region shape. The y-axis has a logarithmic scale. Numbers above bars are the number of virtual instructions executed per dispatch. SPECjvm98 benchmarks appear along X axis sorted by the average number of instructions executed by a LB.

system, execution enters the code cache whenever a region body is called from a dispatch loop. It is an easy matter to instrument the dispatch loops to count how many iterations occur, and hence how many dispatches are made. These numbers are reported by Figure 7.1. The figure shows how direct call threading (DCT) compares to linear blocks (LB), interpreted traces with no linking (i-TR-nolink) and linked interpreted traces (i-TR). Note the y-axis has a logarithmic scale.

DCT dispatches each virtual instruction body individually, so the DCT bars on Figure 7.1 report how many virtual instructions were executed by each benchmark. For each benchmark, the ratio of DCT to LB shows the dynamic average linear block length (e.g., for `compress` the average linear block executed $1.25 \times 10^{10} / 1.27 \times 10^{9} = 9.9$ virtual instructions). In general, the height of each bar on Figure 7.1 divided by the height of the DCT bar gives the average

number of virtual instructions executed per dispatch of that region shape. Figure 7.2 presents the data this way – also, benchmarks are sorted by the average LB length. Hence, for compress, the LB bar shows 9.9 virtual instructions executed on the average.

Scientific benchmarks appear on the right of Figure 7.2 because they tend to have longer linear blocks. For instance, the average block in `scitest` has about 24 virtual instructions whereas `javac`, `jess` and `jack` average about 4 instructions. Comparing the geometric mean across benchmarks, we see that LB reduces the number of dispatches relative to DCT by a factor of 6.3. On long basic block benchmarks, we expect that the performance of LB will approach that of direct threading for two reasons. First, fewer trips around the dispatch loop are required. Second, we showed in Chapter 5 that subroutine threading is better than direct threading for linear regions of code.

Traces do predict paths taken through the program. The rightmost cluster on Figure 7.2 show that, even without trace linking (i-TR-nolink), the average trace executes about 5.7 times more virtual instructions per dispatch than a LB. The improvement can be dramatic. For instance `javac` executes, on average, about 22 virtual instructions per trace dispatch. This is much longer than its dynamic average linear block length of 4 virtual instructions. This means that for `javac`, on the average, the fourth or fifth trace exit is taken. Or, putting it another way, for `javac` a trace typically correctly predicts the destination of 5 or 6 virtual branches.

This behavior confirms the assumptions behind our approach to handling virtual branch instructions in general and the design of interpreted trace exits in particular. We expect that most of the trace exits, four fifths in the case of `javac`, will not exit. Hence, we generate code for interpreted trace exits that should be easily predicted by the processor's branch history predictors. In the next section we will show that this improves performance and in Section 7.5 we show that it also reduces branch mispredictions.

Adding trace linking completes the interpreted trace (i-TR) technique. Trace linking makes the greatest single contribution, reducing the number of times execution leaves the trace cache by between one and 3.7 *orders of magnitude*. Trace linking has so much impact because it

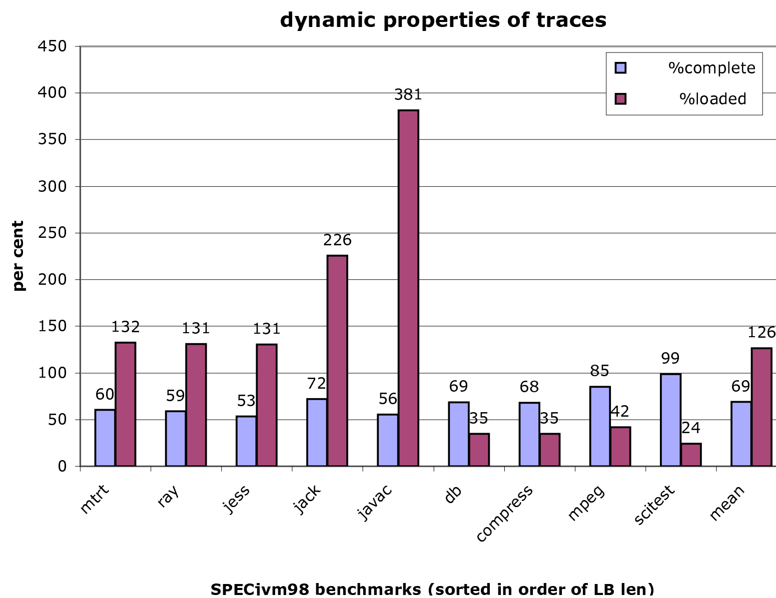


Figure 7.3: Percentage trace completion rate as a proportion of the virtual instructions in a trace and code cache size for as a percentage of the virtual instructions in all loaded methods. For the SPECjvm98 benchmarks and scitest.

links traces together around loops. A detailed discussion of how inner loops depend on trace linking appears in Section 6.4.3.

Although this data shows that execution is overwhelmingly from the trace cache it gives no indication of how effectively code cache memory is being used by the traces. A thorough treatment of this, like the one done by Bruening and Duesterwald [10], is beyond the scope of this thesis. Nevertheless, we can relate a few anecdotes based on data that our profiling system already collects.

Figure 7.3 describes two aspects of traces. First, in the figure, the %complete bars report the extent to which traces typically complete, measured as a percentage of the virtual instructions in a trace. For instance, for `raytrace`, the average trace exit occurs after executing 59% of the virtual instructions in the trace. Second, the %loaded bars report the size of the traces in the code cache as a percentage of the virtual instructions in all the loaded methods. For raytrace we see that the traces contain, in total, 131% of the code in the underlying loaded methods.

We observe that for an entire run of the `scitest` benchmark all generated traces contain

only 24% of the virtual instructions contained in all loaded methods. This is a good result for traces, suggesting that a trace-based JIT needs to compile fewer virtual instructions than a method-based JIT. Also, we see that for `scitest`, the average trace executes almost to completion, exiting after executing 99% of the virtual instructions in the trace. This is what one would expect for a program that is dominated by inner loops with no conditional branches – the typical trace will execute until the reverse branch at its end.

On the other hand, for `javac` we find the reverse, namely that the traces bloat the code cache – almost four *times* as many virtual instructions appear in traces than are contained in the loaded methods. In Section 7.5 we shall discuss the impact of this on the instruction cache. Nevertheless, traces in `javac` are completing only modestly less than the other benchmarks. This suggests that `javac` has many more hot paths than the other benchmarks. What we are not in a position to measure at this point is the temporal distribution of the execution of the hot paths.

todo: javac bloat marc's MSc

7.3 Effect of region shape on performance

In this section we report the elapsed time required to execute each benchmark. One of our main goals is to create an architecture for a high level machine that can be gradually extended from a simple interpreter to a high performance JIT augmented system. Here we evaluate the performance of various stages of Yeti's enhancement from a direct call-threaded interpreter to a trace based mixed-mode system.

Figure 7.4 shows how performance varies as differently shaped regions of the virtual program are executed. The figure shows elapsed time relative to the unmodified JamVM distribution, which uses direct-threaded dispatch. The raw performance of unmodified JamVM and TR-JIT is given in Table 7.1. The first four bars in each cluster represent the same stage of Yeti's enhancement as those in Figure 7.1. The fifth bar, TR-JIT, gives the performance of Yeti with our JIT enabled.

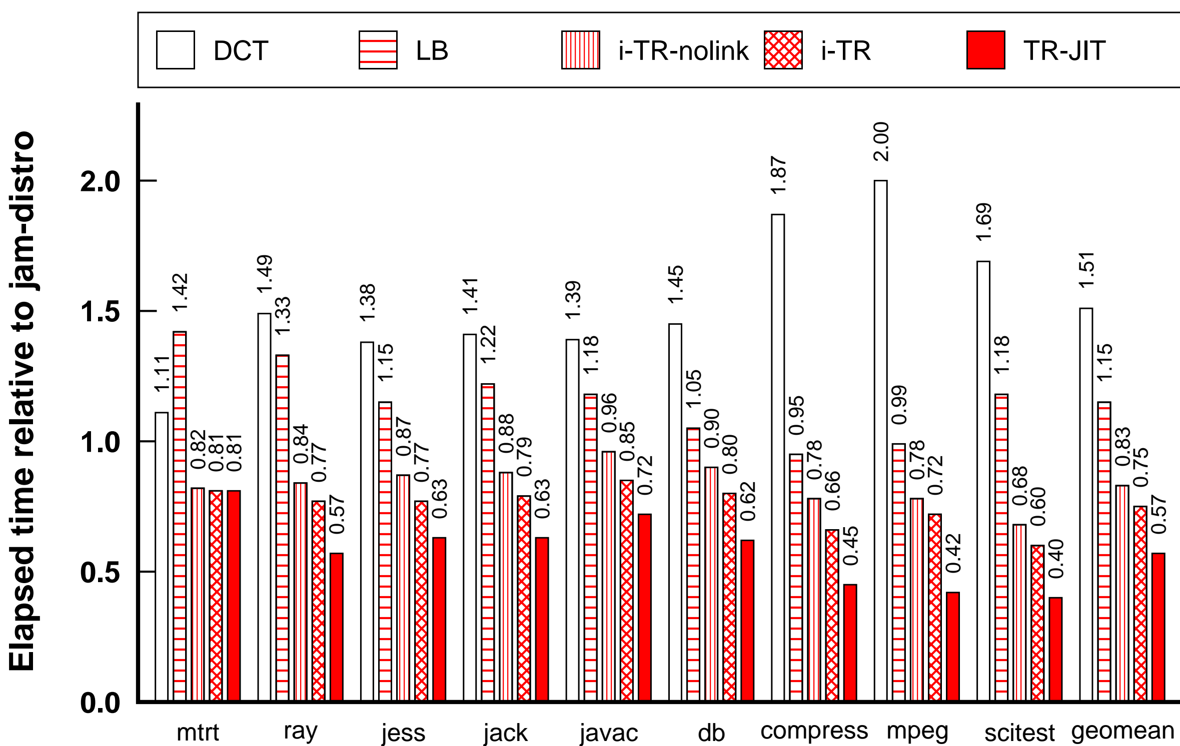


Figure 7.4: Performance of each stage of Yeti enhancement from DCT interpreter to trace-based JIT relative to unmodified JamVM-1.3.3 (direct-threaded) running the SPECjvm98 benchmarks (sorted by LB length).

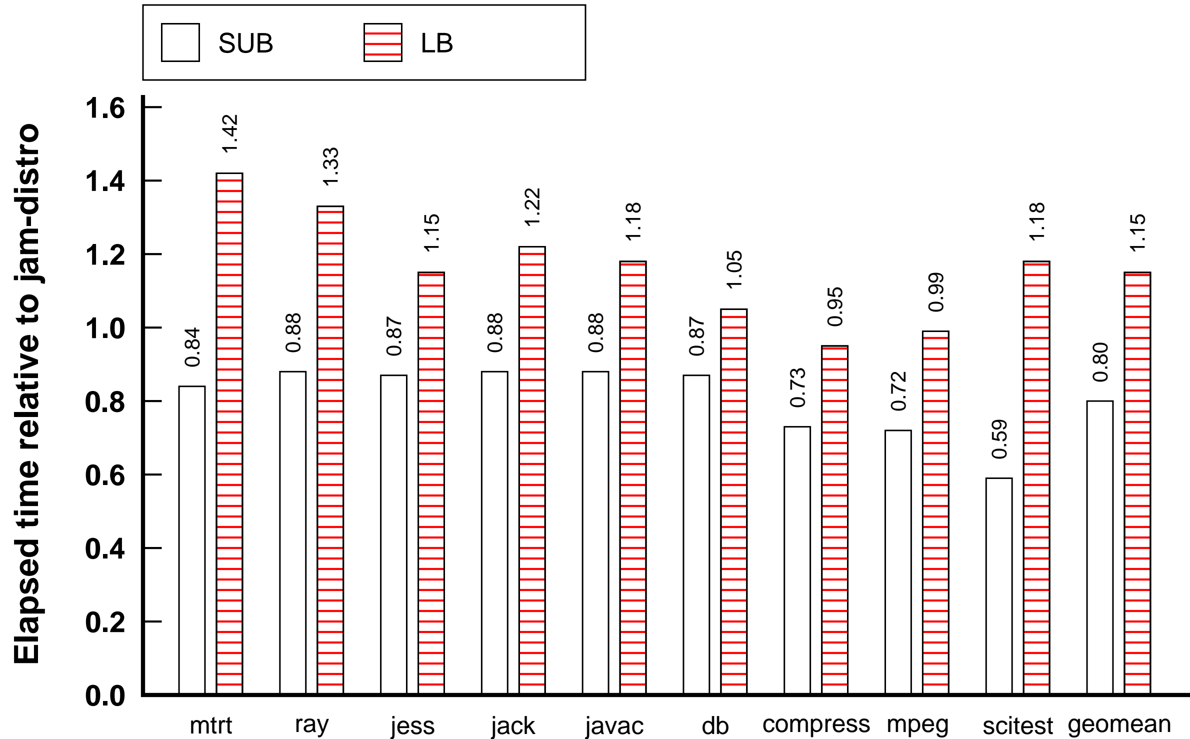


Figure 7.5: Performance of Linear Blocks (LB) compared to subroutine-threaded JamVM-1.3.3 (SUB) relative to unmodified JamVM-1.3.3 (direct-threaded) for the SPECjvm98 benchmarks.

Direct Call Threading Our simplest technique, direct call threading (DCT) is slower than JamVM, as distributed, by about 50%.

Although this seems serious, we note that many production interpreters are not direct threaded but rather use the slower and simpler switch threading technique. When JamVM is configured to run switch threading we we find that its performance is within 1% of DCT. This suggests that the performance of DCT is well within the useful range.

Linear Blocks As can be seen on Figure 7.4, Linear blocks (LB) run roughly 30% faster than DCT, matching the performance of direct threading for benchmarks with long basic blocks like `compress` and `mpeg`. On the average LB runs only 15% more slowly than direct threading.

The region bodies identified at run time by LB are very similar to the code generated by subroutine threading (SUB) at load time so one might expect the performance of the two tech-

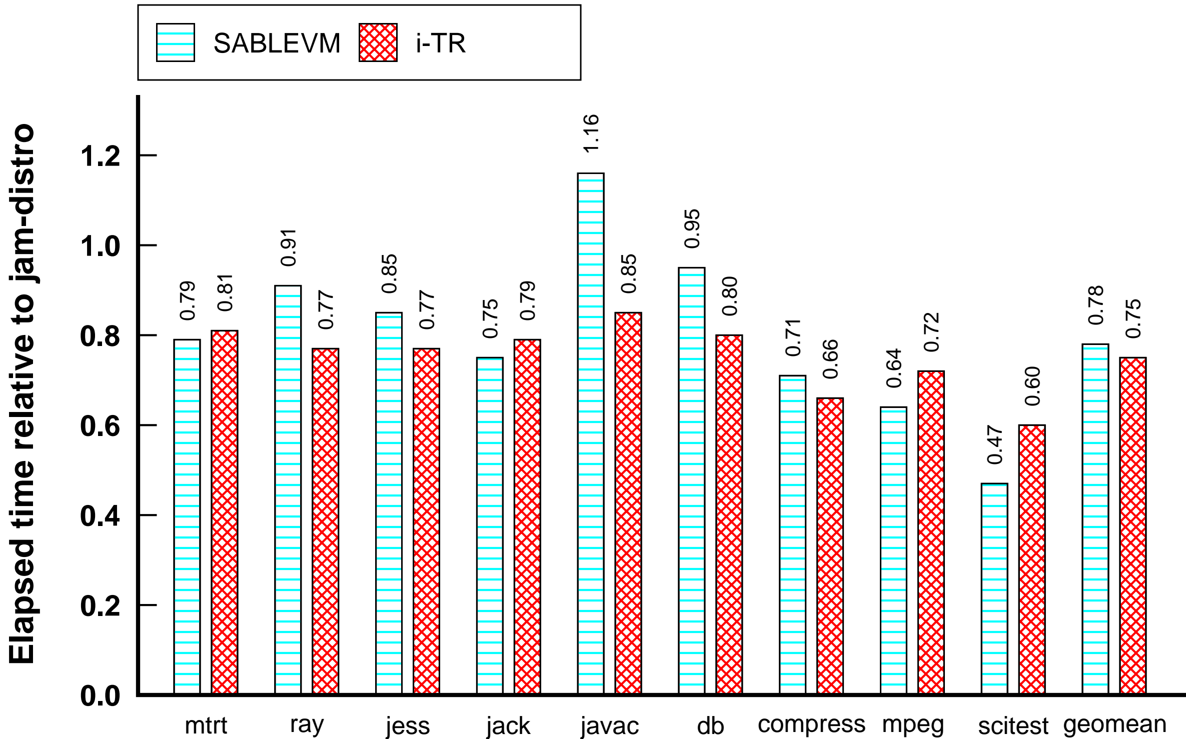


Figure 7.6: Performance of interpreted traces (i-TR) compared to SableVM relative to unmodified JamVM-1.3.3 (direct-threaded) for the SPECjvm98 benchmarks.

niques to be the same. However, as shown by Figure 7.5 LB is, on the average, about 43% slower.

This is because virtual branches are much more expensive for LB. In SUB the virtual branch body is called from the CTT[1], then, instead of returning, it executes an indirect branch directly to the destination CTT slot. In contrast, in LB a virtual branch instruction sets the vPC and returns to the dispatch loop to call the destination region body. In addition each iteration of the dispatch loop must loop up the destination body in the dispatcher structure (through an extra level of indirection compared to SUB).

Interpreted Traces Just as LB reduces dispatch and performs better than DCT, so link-disabled interpreted traces (i-TR-nolink) further reduce dispatch and run 38% faster than LB.

Interpreted traces implement virtual branch instructions better than LB or SUB. As de-

[1]See Section 3.6

scribed in Section 6.4.1, i-TR generates a trace exit for each virtual branch. The trace exit is implemented as a direct conditional branch that is not taken when execution stays on trace. As we have seen in the previous section, execution typically remains on trace for several trace exits. Thus, on the average, i-TR replaces costly indirect calls (from the dispatch loop) with relatively cheap not-taken direct conditional branches. Furthermore, the conditional branches are fully exposed to the branch history prediction facilities of the processor.

Trace linking, though it eliminates many more dispatches, achieves only a modest further speed up because the specialized dispatch loop for traces is much less costly than the generic dispatch loop that runs LB.

Figure 7.6 compares i-TR to selective inlining as implemented by SableVM 1.1.8. SableVM wins on programs with long basic blocks, like `mpeg` and `scitest` because selective inlining eliminates dispatch from long sequences of simple virtual instructions. However, i-TR wins on shorter block programs like `javac` and `jess` by improving branch prediction. Overall, i-TR and SableVM are almost the same.

Subroutine threading again emerges as a very effective interpretation technique, especially given its simplicity. SUB runs only 6% more slowly than i-TR and SableVM.

The fact that i-TR runs exactly the same runtime profiling instrumentation as TR-JIT makes it qualitatively a very different system than SUB or SableVM. SUB and SableVM are both tuned interpreters that generate a small amount of code at load time to optimize dispatch. Neither includes any profiling infrastructure. In contrast to this, i-TR runs all the infrastructure needed to support a JIT. As we shall see in Section 7.5, the improved virtual branch performance of interpreted traces has made it possible to build a profiling system that runs faster than most interpreters.

JIT Compiled traces The rightmost bar in each cluster of Figure 7.4 shows the performance of our best-performing version of Yeti (TR-JIT). Comparing geometric means, we see that TR-JIT is roughly 24% faster than interpreted traces. Despite supporting only 50 integer and

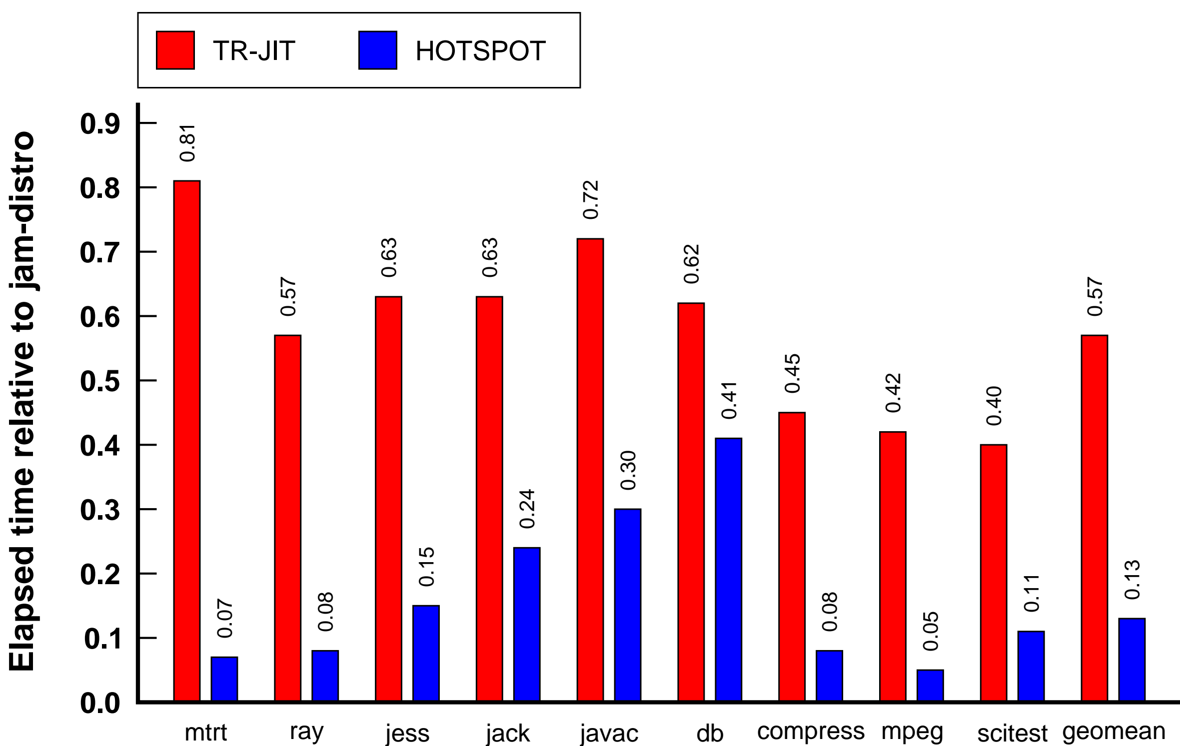


Figure 7.7: Elapsed time performance of Yeti with JIT compared to Sun Java 1.05.0_6_64 relative to JamVM-1.3.3 (direct threading) running SPECjvm98 benchmarks.

object virtual instructions, our trace JIT improves the performance of integer programs such as `compress` significantly. With our most ambitious optimization, of virtual method invocation, TR-JIT improved the performance of `raytrace` by about 35% over i-TR. `Raytrace` is written in an object-oriented style with many small methods invoked to access object fields. Hence, even though it is a floating-point benchmark, it is greatly improved by devirtualizing and inlining these accessor methods.

Figure 7.7 compares the performance of TR-JIT to Sun Microsystems' Java HotSpot JIT. Our current JIT runs the SPECjvm98 benchmarks 4.3 times slower than HotSpot. Results range from 1.5 times slower for `db`, to 12 times slower for `mtrt`. Not surprisingly, we do worse on floating-point intensive benchmarks since we do not yet compile the float bytecodes.

7.4 Early Pentium Results

As illustrated earlier, in Figure 3.4, the Intel's Pentium architecture takes a different approach to indirect branches and calls than does the PowerPC. On the PowerPC we have shown that the two-part indirect call used in Yeti's dispatch loops a performs well. However, the Pentium relies on its BTB to predict the destination of its indirect call instruction. As we saw in Chapter 5, when the prediction is wrong many stall cycles may result. Conceivably, on the Pentium, the unpredictability of the dispatch loop indirect call could lead to very poor performance.

Gennady Pekhimenko, a fellow graduate student at the University of Toronto, ported i-TR to the Pentium platform. Figure 7.8 gives the performance of his prototype. The results are roughly comparable to our PowerPC results, though i-TR outperforms direct threading a little less on the Pentium. The average test case ran in 83% of the time taken by direct threading whereas it needed 75% on the PowerPC.

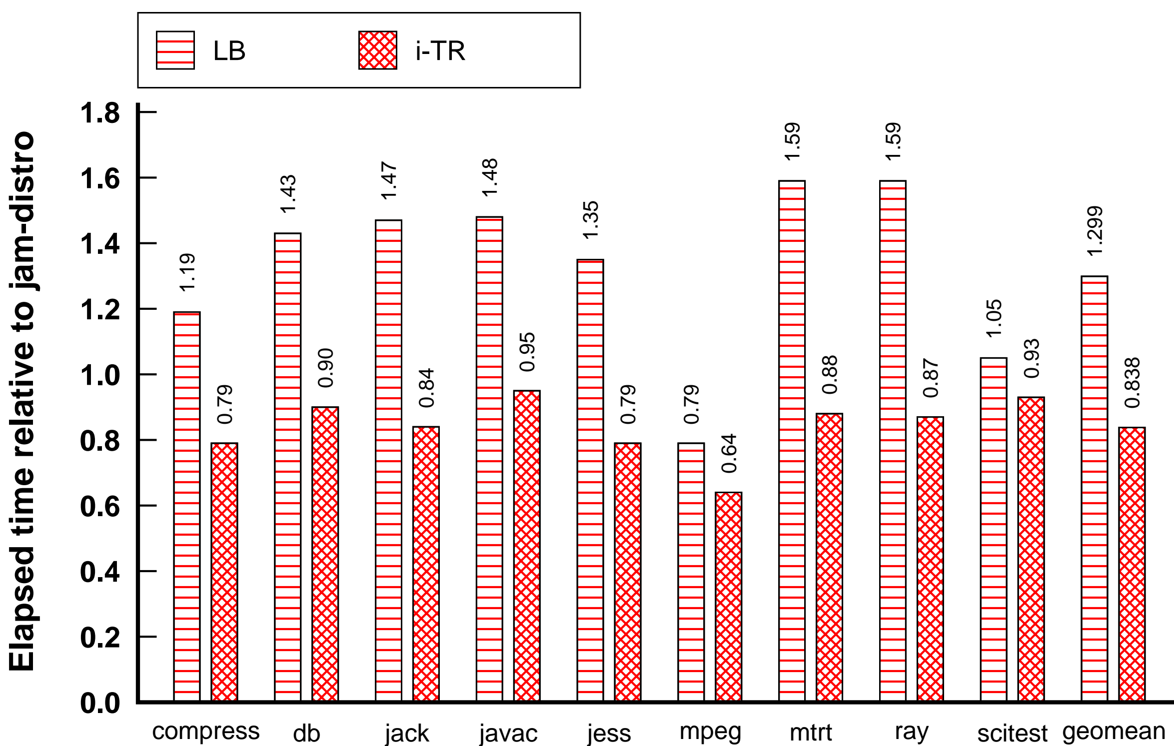


Figure 7.8: Performance of Gennady Pekhimenko's Pentium port relative to unmodified JamVM-1.3.3 (direct-threaded) running the SPECjvm98 benchmarks.

7.5 Identification of Stall Cycles

We have shown that Yeti performs well compared to existing interpreter techniques. However, much of our design is motivated by micro-architectural considerations. In this section we use a new set of tools to measure the stall cycles experienced by Yeti as it runs.

The purpose of this analysis is twofold. First, we would like to confirm that we understand why Yeti performs well. Second, we would like to discover any source of stalls we did not anticipate, and perhaps find some guidance on how we could do better.

7.5.1 GPUL

Azimi et al [40] describe a system that uses a statistical heuristic to attribute stall cycles in a PowerPC 970 processor. They define a *stall cycle* as a cycle for which there is no instruction that can be completed. Practically speaking, on a PowerPC970, this occurs when the processor's completion queue is empty because instructions are held up, or stalled. Their approach, implemented for a PPC970 processor running K42, a research operating system [17], exploits performance monitoring hardware in the PowerPC that recognizes when the processor's instruction completion queue is empty. Then, the next time an instruction *does* complete they attribute, heuristically and imperfectly, all the intervening stall cycles to the functional unit of the completed instruction. Azimi shows statistically that their heuristic estimates the true causes of stall cycles well. Recently, Livio Soares, a member of the same research group as Azimi, created a tool called GPUL, which adds similar performance monitoring to the 2.6.18 Linux kernel.

GPUL runs only on Linux and at the moment only works on a PowerPC 970FX processor[2]. This is slightly different than the PowerPC 970 processor we have been using up to this point. The only acceptable machine we have access to is an Apple Xserve system which was also slightly faster than our machine, running at 2.3 GHz rather than 2.0 GHz.

[2]We suspect that the actual requirement is the interrupt controller that Apple packages in newer systems.

category name	Description
i-cache	Instruction cache misses
br_misp	Branch mispredictions
compl	Completed instructions. (Cycles in which an instruction did complete)
other_stall	Miscellaneous stalls
fxu	Fixed point execution unit
fpu	Floating point execution unit
d-cache	Data cache
basic_lsu	Basic load and store unit stalls

Table 7.3: GPUL categories

7.5.2 GPUL results

Figure 7.9 shows the results of the GPUL tools to break down stall cycles for various runs of the SPECjvm98 benchmarks.

Five bars appear for each benchmark. From the left to the right, the stacked bars represent subroutine-threaded JamVM 1.1.3 (SUB) , JamVM 1.1.3 (direct-threaded as distributed, hence DISTRO) and three configurations of Yeti, i-TR-no-link, i-TR and TR-JIT. The y axis, like many of our performance graphs, reports performance relative to JamVM. The height of the DISTRO bar is thus 1.0 by definition. Figure 7.10 reports the same data as Figure 7.9, but, in order to facilitate pointing out specific trends, zooms in on four specific benchmarks.

Each histogram column is split vertically into a stack of bars which illustrates how executed cycles break down by category. Only cycles listed as “compl” represent cycles in which an instruction completed. All the other categories represent stalls, or cycles in which the processor was unable to complete an instruction. The “other_stall” category represents stalls to which the tool was not able to attribute a cause. Unfortunately, the other_stall category includes a source of stalls that is important to our discussion, namely the stalls caused by data dependency be-

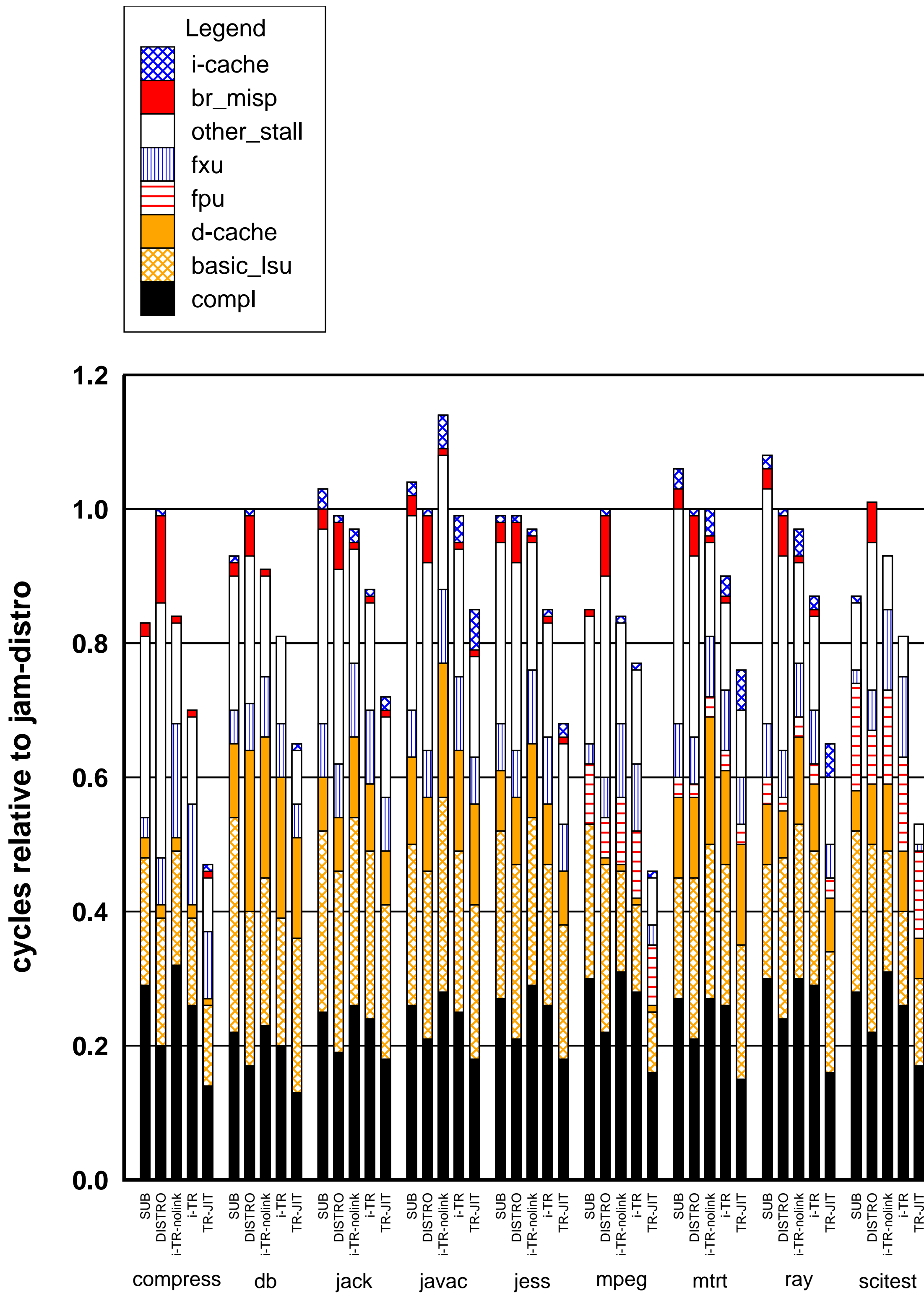


Figure 7.9: GPUL relative to JamVM-1.3.3 (direct threading) running SPECjvm98 benchmarks.

tween the two instructions of the PowerPC architectures' two-part indirect branch mechanism[3]. See Figure 3.4 for an illustration of two-part branches.

todo:m maybe new TE figure for i-TR?

The total cycles executed by each benchmark, as reported by GPUL, do not correlate perfectly with the elapsed time measurements reported earlier in this chapter.

For instance, in Figure 7.4, i-TR runs scitest in 60% of the time of direct threading, whereas in Figure 7.10(c) it takes 80%. There are a few important differences between the runs, namely the differences between the PowerPC 970FX and PowerPC 970, the different clock speed (2.3 GHz vs 2.0 GHz) and differences between Linux (with GPUL modifications) and OSX 10.4. We use the GPUL data qualitatively to characterize pipeline hazards and not to measure absolute performance.

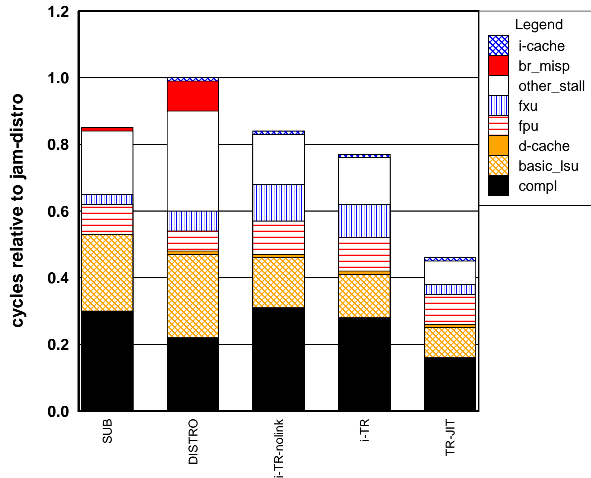
7.5.3 Trends

Several interesting trends emerge from our examination of the GPUL reports.

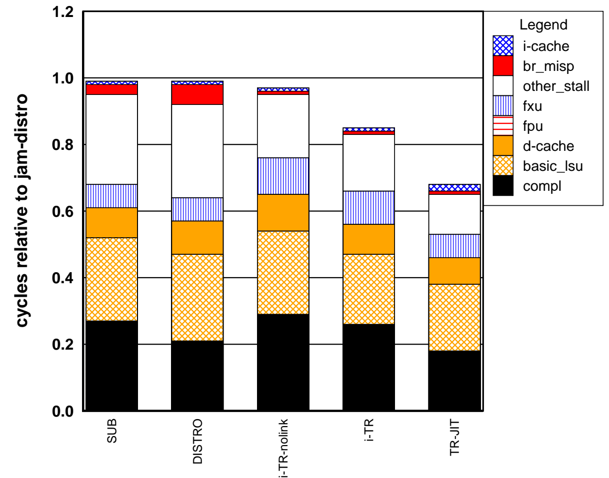
1. Interpreted traces reduce branch mispredictions caused by virtual branch instructions.
2. Simple code we generated for interpreted trace exits stresses the fixed-point execution unit (fxu)
3. Our JIT (TR-JIT) does little to reduce lsu stalls, which is a surprise since many loads and stores to the expression stack are eliminated by the register allocator.
4. As we reduce pipeline hazards caused by dispatch new kinds of stalls arise.
5. Trace bloat, like we observed for javac, can lead to significant stalls due to instruction cache misses.

Each of these issues will be discussed in turn.

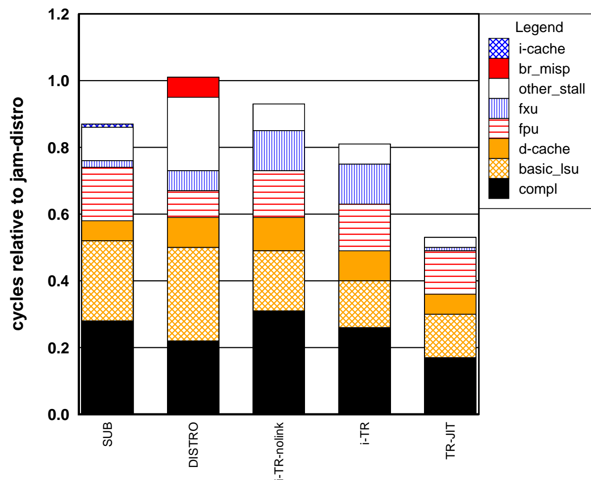
[3] In earlier models of the PowerPC, for instance the 7410, these cycles were called "LR/CTR stall cycles", as reported by Figure 5.1(b)



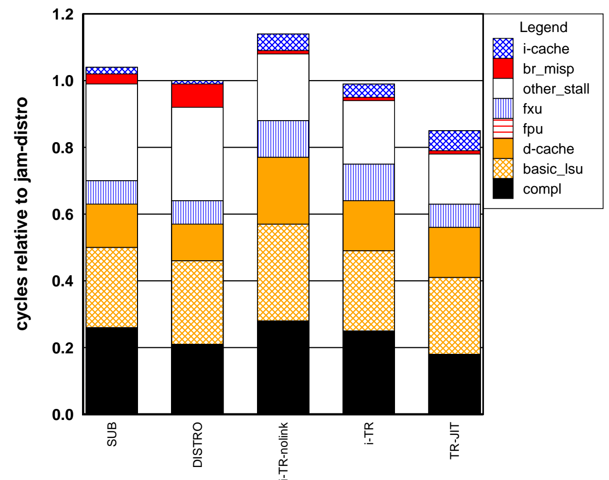
(mpeg) – long int blocks



(jess) – short blocks



(scitest) – long float blocks



(javac) – trace cache bloat

Figure 7.10: stall breakdown for SPECjvm98 benchmarks relative to JamVM-1.3.3 (direct threading).

Branch misprediction

In Figure 7.10(mpeg) we see how our techniques affect `mpeg`, which has a few very hot, very long basic blocks. The blocks contain many duplicate virtual instructions. Hence, direct threading encounters difficulty due to the context problem, as discussed in Section 3.5. (This is plainly evident in the solid red br_misp stack on the DISTRO bar on all four sub figures.)

SUB reduces the mispredictions that occur running `mpeg` significantly – presumably the ones caused by linear regions. Yeti's i-TR technique effectively eliminates the branch mispredictions for `mpeg` altogether. Both techniques also reduce other_stall cycles relative to direct threading. These are probably being caused by the PowerPC's two-part indirect branches which are used by DISTRO to dispatch all virtual instructions and by SUB to dispatch virtual branches. SUB eliminates the delays for straight-line code and i-TR further eliminates the stalls for virtual branches. Figures 7.10(javac) and 7.10(jess) show that traces cannot predict all branches and some stalls due to branch mispredictions remain for i-TR and TR-JIT.

Overhead of interpreted Trace Exits

In all four sub figures of Figure 7.10 we see that fxu stalls decrease or stay the same relative to DISTRO for SUB whereas for i-TR they increase. Note also that the fxu stalls decrease again for the TR-JIT condition. This suggests that the fxu stalls are not caused by the overhead of profiling (since TR-JIT runs exactly the same instrumentation as i-TR). Rather, they are caused by the overhead of the simple-minded trace exit code we generate for interpreted traces.

Recall that interpreted traces generate a compare immediate of the `vPC` followed by a conditional branch. The comparand is the destination `vPC`, a 32 bit number. On a PowerPC there is no form of the compare immediate instruction that takes a 32 bit immediate parameter. Thus, we generate two fixed point load immediate instructions to load the immediate argument into a register. Presumably it is these fixed point instructions that are causing the extra stalls.

TR-JIT and the Expression Stack

Yeti's compiler works hard to eliminate loads and stores to and from Java's expression stack. In Figure 7.10(mpeg) , TR-JIT makes a large improvement over i-TR by reducing the number of completed instructions. However, it was surprising to learn that basic_lsu stalls were in fact not much effected. (This pattern holds across all the other sub figures also.) Presumably the pops from the expression stack hit the matching pushes in PPC970's store pending queue and hence were not stalling in the first place.

Exposing stalls from workload

In Figure 7.10(scitest) we see an increase in stalls due to the FPU for SUB and i-TR. Our infrastructure makes no use of the FPU at all – so presumably this happens because stalls waiting on the real work of the application are exposed as we eliminate other pipeline hazards.

This effect makes it hard to draw conclusions about any increase in stalls that occurs, for instance the increase in fxu stalls caused by i-TR described in the previous section, because it might also be caused by the application.

Trace bloat

The `javac` compiler is a big benchmark. The growth of the blue hatched bars at the top of Figure 7.10(javac) shows how i-TR and TR-JIT make this significantly worse. Even SUB, which only generates one additional 4 byte call per virtual instruction, increases i-cache misses. In the figure, i-TR stalls on instruction cache as much as direct threading stalls on mispredicted branches.

As we pointed out in Section 7.2, Dynamo's trace selection heuristic does not work well for `javac`, selecting traces representing eight times as many virtual instructions as appear in all the loaded methods. This happens when many long but slightly different paths are hot through a body of code. Part of the problem is that the probability of reaching the end of a long trace under these conditions is very low. As trace exits become hot more traces are generated and

replicate even more code. As more traces are generated the trace cache grows huge.

Figure 7.10(javac) shows that simply setting aside a large trace cache is not a good solution. The replicated code in the traces makes the working set of the program larger than the instruction cache can hold.

Our system does not, at the moment, implement any mechanism for reclaiming memory that has been assigned to a region body. An obvious candidate would be reactive flushing (See Section 2.5), which occasionally flushes the trace cache entirely. This may result in better locality of reference after the traces are regenerated anew. Counter-intuitively, reducing the size of the trace cache and implementing a very simple trace cache flushing heuristic may lead to better instruction cache behavior than setting aside a large trace cache.

Hiniker et al [40] have suggested several changes to the trace selection heuristic that improve locality and reduce replication between traces.

7.6 Chapter Summary

We have shown that traces, and especially linked traces, are an effective shape for region bodies. The trace selection heuristic described by the HP Dynamo project, described in Section 2.5, results in execution from the code cache for an average or 2000 virtual instructions between dispatches. This reduces the overhead of region body dispatch to a negligible level. The amount of code cache memory required to achieve this seems to vary widely by program, from a very parsimonious 24% of the virtual instructions in the loaded methods for `scitest` to a rather bloated 380% for `javac`.

We have measured the performance of four stages in the evolution of Yeti: DCT, LB, i-TR, and TR-JIT. Performance has steadily improved as larger region bodies are identified and translated. Traces have proven to be an effective shape for region bodies for two main reasons. First, interpreted traces offer a simple and efficient way to efficiently dispatch both straight line code and virtual branch instructions. Second, compiling traces is straightforward – in part

because the JIT can fall back on our callable virtual instruction bodies, but also because traces contain no merge points, which makes compilation easy.

Yeti provides a design trajectory by which a high level language virtual machine can be extended from a simple interpreter to a sophisticated trace-based JIT compiler mixed-mode virtual machine. Our strategy is based on two key assumptions. First, that stepping back to a relatively slow dispatch technique, direct call threading (DCT), is worthwhile. Second, that identifying dynamic regions of the program at runtime, traces, should be done early in the life of a system because it enables high performance interpretation.

In this chapter we have shown that both these assumptions are reasonable. Our implementation of DCT performs no worse than switch threading, commonly used in production, and the combination of trace profiling and interpreted traces is competitive with high-performance interpreter optimizations. This is in contrast to context threading, selective inlining, and other dispatch optimizations, which perform about the same as interpreted traces but do nothing to facilitate the development of a JIT compiler.

A significant remaining challenge is how best to implement callable virtual instruction bodies. The approach we follow, as illustrated by Figure 4.2, is efficient but depends on C language extensions and hides the true control flow of the interpreter from the compiler that builds it. A possible solution to this will be touched upon in Chapter 8.

The GPUL infrastructure has enabled us to learn why our technique does well. As expected, we find that traces make it easier for the branch prediction hardware to do its job, and thus stalls due to branch mispredictions reduce markedly. To be sure, some paths are still hard to predict and traces do not eliminate all mispredicted branches. We find that the extra path length of interpreted trace exits does matter, but in the balance reduces stall cycles from mispredicted branches more than enough to improve performance overall.

Yeti is early in its evolution at this point. Given the robust performance increases we obtained compiling the first 50 integer instructions we believe much more performance can be easily obtained just by compiling more kinds of virtual instructions. For instance, floating

point multiplication (FMUL or DMUL) appears amongst the most frequently executed virtual instructions in four benchmarks (`scitest`, `ray`, `mpeg` and `mtrt`). We expect that our gradual approach will allow these virtual instructions to compiled next with commensurate performance gains.

Chapter 8

Conclusions and Future Work

8.1 Conclusions and Lessons Learned

Interpreters play an important role in the implementation of computer languages. Initially, language implementors need a language VM to be simple and flexible in order to support the evolution of their language. Later, as their language increases in popularity, performance may become more of a concern.

Today, commonly implemented interpreter designs do not anticipate the need for more performance, and just in time (JIT) compiler designs, though capable of very high performance, require a great deal of up-front development. These factors conspire to prevent, or at least delay, important language implementations from improving performance by deploying a JIT. In this thesis we have responded to this challenge by describing a design for a language VM that explicitly maps out a trajectory of staged deployments, providing gradually increasing performance as development effort is invested.

Our approach is novel in several ways.

1. We package virtual instruction bodies as callable, and dispatch using direct call threading (DCT). DCT runs about the same speed as switch treading. Thus, although it is slower than some dispatch techniques DCT performs well enough to be useful. Virtual instruc-

tions have been implemented as callable in the past, but the technique fell out of use because the path length of call and return was costly. We show that callable bodies can be very efficient now that processors commonly implement return branch predictors.

2. We realized that although the overhead of DCT is high for dispatching single virtual instruction bodies, it may be perfectly reasonable for dispatching callable region bodies generated from dozens or hundreds of virtual instructions. The basic idea behind Yeti's extensibility is that development effort should be invested in identifying and compiling larger and more complex regions of the virtual program which are then dispatched from a DCT loop.

3. Optimizing the dispatch of virtual branch instructions is typically done when a method is loaded. Instead, we identify traces at run time using profiling instrumentation called from the dispatch loop. Hot traces predict paths through the virtual program which we exploit to generate simple trace exit code in otherwise subroutine-threaded interpreted traces. These perform well, resulting in as good performance as as SableVM's selective inlining or our own branch-inlining.

4. When even better performance is needed, we show how a trace-based JIT can be built to eliminate dispatch and replace the expression stack with register-to-register compiled code. The novel aspect of our JIT is that it exploits the fact that Yeti's virtual instruction bodies are callable. Unsupported virtual instructions, or difficult compiler corner cases can be side-stepped by dispatching virtual instruction bodies instead. This allows support for virtual instructions to be added one at a time. The importance of the latter point is hard to quantify, but seemed to reduce the difficulty of debugging the back end of the compiler significantly.

Most of the elements of our approach are plausible as soon as it has been proved that callable bodies can be efficiently dispatched. However, actual performance improvements depend on a subtle trade-off between the overhead of runtime profiling and the reduction of stalls caused

by branch mispredictions. The only way to determine that our ideas were viable is to build a fairly complete prototype. We chose to build a prototype in Java because there are commonly accepted benchmark programs to measure and many high quality implementations to compare ourselves to.

In the process we learned a number of interesting things:

1. Calling virtual instruction bodies is a good approach on modern CPUs. Our implementation of subroutine threading (SUB) is very simple and eliminates most of the branch mispredictions caused by switch or direct threading, particularly those caused by dispatching straight-line code. However, SUB does not address mispredictions caused by dispatching virtual branch instructions. Also, it is difficult to interpose runtime instrumentation into subroutine threaded execution.

2. Branch inlining, our straight-forward approach to improving the virtual branch performance of SUB, is labor intensive and non-portable. It improves the performance of subroutine threading by about 5%.

3. DCT is even simpler than SUB and does not perform any worse than switch. DCT is very easy to augment with profiling, since instrumentation can simply be called from the dispatch loop before and after dispatching each body. Furthermore, by providing multiple dispatch loops it is easy to turn instrumentation on and off.

4. Our trace compiler was easy to build, and we attribute this primarily to two factors. First, traces contain no merge points, so it is easy to track where expression temporary values are on the expression stack and assign them to registers. Second, callable virtual instruction bodies enabled us to add compiler support for virtual instructions one at a time.

The primary weakness of our prototype is the specific mechanism we used to implement callable virtual instruction bodies. Our approach, as illustrated by Figure 4.2, hides the return branch from the compiler. This means that the optimizer does not properly understand

the control flow graph of the interpreter. The workaround, suitable only for a prototype, is to "fake" the missing control flow by adding computed goto statements that are never executed immediately following each inline return instruction. Nested functions, a relatively commonly implemented extension to C, are a promising alternative that will be discussed in the next section.

8.2 Future work

Substantial additional performance gains are no doubt possible by extending our trace-based JIT to handle more types of instructions (such as the floating point bytecodes) and by applying classical optimizations such as common subexpression elimination. Improving the performance of compiled code by applying classical optimizations is relatively well understood. Hence, on its own, such an effort seems to have relatively little to contribute to research. Moreover, it would require significant engineering work and likely can only be undertaken by a well-funded project.

We will discuss four avenues for further research. First, a way to package virtual instruction bodies as nested functions. Second, how the approach we describe in Section 6.4.3 to optimize virtual method invocation could be adapted for dynamically typed languages. Third, we comment on how new shapes of region bodies could be derived from linked traces. Fourth, we describe our vision of how our design could be used by the implementors of a new language.

8.2.1 Virtual instruction bodies as nested functions

An better option for implementing callable virtual instruction bodies might be to define them as nested functions. Nested functions are a common extension to C, implemented by gcc and other C compilers, that allows one function to be declared within another. The idea is that each virtual instruction body is declared as a separate nested function, with all bodies nested within the main interpreter function. Important interpreter variables, like the `vPC`, are defined,

as currently, as local variables in the main interpreter function but can be used from the nested function implementing each virtual instruction body as well.

The approach is elegant, since functions are a natural way to express virtual instruction bodies, and also well supported by the tool chain, including the debugger. However, our first attempts in this direction did not perform well. In short, when a nested function is called via a function pointer, like from our DCT dispatch loop, gcc adds an extra level of indirection and calls the nested function via a runtime generated trampoline. As a result the DCT dispatch loop runs very slowly.

We investigated the possible performance of nested functions by hand-modifying the assembler generated by gcc to short-circuit the trampoline. In this way, we created a one-off version of OCaml that declares each virtual instruction body in its own nested function and runs a simple DCT dispatch loop like the one illustrated by Figure 3.2. On the PowerPC this DCT interpreter runs the same OCaml benchmarks used in Chapter 5 about 22% more slowly than switch threading.

Further improvements to nested function performance should be investigated, possibly including modifications to gcc to create a variant of nested functions more suitable for implementing virtual instruction bodies.

8.2.2 Extension to Dynamically Typed Languages

An exciting possibility is to create new speculative dynamic optimizations based on the runtime profile data collected while training a trace (See Section 6.2.3.) The basic realization is that a mechanism very similar to a trace exit can be used to guard almost any speculative optimization. As a specific example we consider the optimization of arithmetic operations in a dynamically typed language.

A dynamically typed language is a language that does not force the user to declare the types of variables but instead discovers types at run time. A typical implementation compiles expressions to sequences of virtual instructions that are not type specific. For instance, in Tcl

or Python the virtual body for the addition will work for integers, floating point numbers or even strings. Performance tends to be poor as each virtual instruction body must check the type of each input before actually calculating the answer.

We believe the same profiling infrastructure that we use to optimize callsites in Java (Section 6.4.3) could be used to improve arithmetic bytecodes in a dynamically typed language. Whereas the destination of a Java method invocation depends only upon the type of the invoked-upon object, the operation carried out by a polymorphic virtual instruction may depend on the type of *each* input. For instance, suppose that a specific instance of a virtual addition virtual instruction in Tcl, Python or JavaScript has integer type. (We would know this if its inputs were observed to be integers during trace training.) We could generate one or more trace exits , or guards, to ensure that the inputs are actually integers. Following the guards we could generate specialized integer code, or dispatch a version of the addition virtual instruction body specialized for integers.

8.2.3 New shapes of region body

Just as basic blocks are collected into traces, so traces could be collected into yet larger regions for optimization. An obvious possibility would be to identify loop nests amongst the linked traces, and use these as a higher level unit of compilation.

The data recorded by our trace region payload structures already includes the information necessary to build a flowgraph of the program in the code cache. It remains to adapt classical flow graph algorithms to detect nested loops and create a strategy for compiling the resulting code.

There seems to be little point, however, in detecting loop nests without any capability of optimizing them. Thus, this extension of our work should be carried out in a system that includes an optimizer.

8.2.4 Vision for new language implementation

Our vision for a new language implementation would be to start by building a direct call threaded interpreter. Until the issues with nested functions have been dealt with, the virtual bodies would have to be packaged as we described in Chapter 6. The level of performance would be roughly the same as a switch-threaded interpreter.

Then, as more performance is called for, we would add linear blocks, interpreted traces, and trace linking. It would be natural to make these extensions in separate releases of our implementation. We believe that much of the runtime profiling infrastructure we built for Yeti could be reused as is. Finally, when performance requirements demand a JIT compiler could be built. Like Yeti, the first implementation could compile only a subset of the virtual instructions, perhaps only the ones needed to address specific performance issues with a given application.

8.3 Elevator pitch

We have described a design trajectory which enables a high level language virtual machine to be deployed in a sequence of stages, starting with a simple entry-level direct call threaded interpreter, followed by interpreted traces and finally a trace-based just in time compiler. By adopting this approach it would be relatively clear how to make future releases of the language perform better. Our hope is that adoption of our approach will lead to better performing computer language implementations for more users.

Bibliography

[1] Ocaml. http://www.ocaml.org.

[2] The Java hotspot virtual machine, v1.4.1, technical white paper. 2002.

[3] Eric Allman. A conversation with james gosling. *ACM Queue Magazine*, 2(5), July/August 2004.

[4] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, VC Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. In *IBM Systems Journals, Java Performance Issue*, 2000.

[5] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, 1996. Available from: `http://citeseer.nj.nec.com/auslander96fast.html`.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report, Hewlett Packard, 1999. Available from: `http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html`.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the ACM SIGPLAN 2000 Conf. on Prog. Language Design and Impl.*, pages 1–12, Jun. 2000.

[8] Iris Baron. *Dynamic Optimization of Interpreters using DynamoRIO*. PhD thesis, MIT, 2003. Available from: `http://www.cag.csail.mit.edu/rio/iris-sm-thesis.pdf`.

[9] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proc. of the 3rd Intl. Symp. on Code Generation and Optimization*, pages 15–26, Mar. 2005.

[10] Derek Bruening and Evelyn Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *Proc. of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000. Available from: `http://www.eecs.harvard.edu/fddo/papers/108.ps`.

[11] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2000.

[12] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the 1st Intl. Symp. on Code Generation and Optimization*, pages 265–275, Mar. 2003. Available from: `http://www.cag.lcs.mit.edu/dynamorio/CGO03.pdf`.

[13] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml*. O'Reilly France, 2000.

[14] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1988.

[15] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *Proc. of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000. Available from: `http://www.cs.washington.edu/homes/lerns/mojo.pdf`.

[16] Randy Clark and Stephen Koehler. *The UCSD Pascal Handbook*. Prentice-Hall, 1982.

[17] IBM Corporation. K42 research operating system [online]. 2006. Available from: `http://www.research.ibm.com/k42`.

[18] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997. Available from: `http://ieeexplore.ieee.org/iel1/40/12908/00591653.pdf`.

[19] Charles Curley. Life in the FastForth lane. *Forth Dimensions*, 14(4), January-February 1993.

[20] Charles Curley. Optimizing in a BSR/JSR threaded forth. *Forth Dimensions*, 14(5), March-April 1993.

[21] Ron Cytron, Jean Ferrante, B. K. Rosen, M. N Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[22] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the 1st Intl. Symp. on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[23] Peter L. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, Jan. 1984.

[24] Karel Driesen. *Efficient Polymorphic Calls*. Klumer Academic Publishers, 2001.

[25] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *ACM SIGPLAN Notices*, 35(11):202–211, 2000.

[26] M. Anton Ertl. Stack caching for interpreters. In *Proc. of the ACM SIGPLAN 1995 Conf. on Prog. Language Design and Impl.*, pages 315–327, June 1995. Available from: `http://www.complang.tuwien.ac.at/papers/ertl95pldi.ps.gz`.

[27] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. *Lecture Notes in Computer Science*, 2150, 2001.

[28] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proc. of the ACM SIGPLAN 2003 Conf. on Prog. Language Design and Impl.*, pages 278–288, June 2003.

[29] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. VMgen — a generator of efficient virtual machine interpreters. *Software Practice and Experience*, 32:265–294, 2002.

[30] S. Fink and F. Qian. Design, implementation, and evaluation of adaptive recompilation with on-stack replacement. In *In Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 2003. Available from: `http://www.research.ibm.com/people/s/sfink/papers/cgo03.ps.gz`.

[31] Etienne Gagnon and Laurie Hendren. Effective inline threading of Java bytecode using preparation sequences. In *Proc. of the 12th Intl. Conf. on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 170–184. Springer, Apr. 2003.

[32] Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proc. of the 2nd Intl. Conf. on Virtual Execution Environments*, pages 144–153, 2006.

[33] Stephen Gilmore. Programming in standard ML '97: A tutorial introduction. 1997. Available from: `http://www.dcs.ed.ac.uk/home/stg`.

[34] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[35] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, 1983.

[36] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[37] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan.J. Eggers. An evaluation of staged run-time optimizations in Dyc. In *Conference on Programming Language Design and Implementation*, May 1999. Available from: `http://www.cs.washington.edu/research/projects/unisw/DynComp/www/Papers%/pldi99.pdf`.

[38] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, Nov. 2001.

[39] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[40] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *Proc. of the 38th Intl. Symp. on Microarchitecture*, pages 141–154, Nov. 2005.

[41] Glenn Hinton, Dave Sagar, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1, 2001. Available from: `http://www.intel.com/technology/itj/q12001.htm`.

[42] Urs Hölzle. *Adaptive Optimization For Self:Reconciling High Performance With Exploratory Programming*. PhD thesis, Stanford University, 1994.

[43] Urs Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Conference on Programming Language Design and Implementation*, 1992. Available from: `http://www.cs.ucsb.edu/labs/oocsb/papers/pldi92.pdf`.

[44] Urs Hölzle and David Ungar. A third-generation Self implementation: Reconciling responsiveness with performance. In *Proceedings of the OOPSLA '94 conference on Object Oriented Programming Systems Languages and Applications*, 1994. Available from: `http://research.sun.com/self/papers/third-generation.html`.

[45] IBM Corporation. *IBM PowerPC 970FX RISC Microprocessor, version 1.6*. 2005.

[46] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*. 2004.

[47] Ronald L. Johnston. The dynamic incremental compiler of apl 3000. In *Proceedings of the international conference on APL: part 1*, pages 82–87, 1979. Available from: `http://doi.acm.org/10.1145/800136.804442`.

[48] Thompson K. Regular expression search algorithm. *CACM*, June 1968.

[49] Peter M. Kogge. An architectural trail to threaded- code systems. *IEEE Computer*, 15(3), March 1982.

[50] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, 1996.

[51] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[52] Robert Lougher. JamVM [online]. Available from: `http://jamvm.sourceforge.net/`.

[53] Motorola Corporation. *MPC7410/MPC7400 RISC Microprocessor User's Manual, Rev. 1*. 2002.

[54] Steven S Muchnick. *Advanced Compiler Design and Construction*. Morgan Kaufman, 1997.

[55] Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proc. of the 16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 195–210, Oct. 2001. Available from: `http://www.cs.nyu.edu/phd_students/pechtcha/pubs/oopsla01.pdf`.

[56] Rob Pike, Bart Locanthi, and John Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software - Practice and Experience*, 15(2):131–151, 1985. Available from: `http://citeseer.nj.nec.com/324101.html`.

[57] Ian Piumarta. Ccg: A tool for writing dynamic code generators. In *OOPSLA'99 Workshop on simplicity, performance and portability in virtual machine design*, Nov. 1999. Available from: `http://piumarta.com/ccg`.

[58] Ian Piumarta. The virtual processor: Fast, architecture-neutral dynamic code generation. In *2004 USENIX Java Virtual Machine Symposium*, 2004.

[59] Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. In *Proc. of the ACM SIGPLAN 1998 Conf. on Prog. Language Design and Impl.*, pages 291–300, June 1998.

[60] R. Pozo and B. Miller. *SciMark: a numerical benchmark for Java and C/C++.*, 1998. Available from: `http://www.math.nist.gov/SciMark`.

[61] Brad Rodriguez. Benchmarks and case studies of forth kernels. *The Computer Journal*, 60, 1993.

[62] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proc. ASPLOS 7*, pages 150–159, October 1996.

[63] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Helsinki University Faculty of Information Technology, May 1996.

[64] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE-COMPUTER*, 38(5):32–38, May 2005.

[65] SPECjvm98 benchmarks [online]. 1998. Available from: `http://www.spec.org/osg/jvm98/`.

[66] Kevin Stoodley. Productivity and performance: Future directions in compilers [online]. 2006. Available from: `http://www.cgo.org/cgo2006/html/StoodleyKeynote.ppt`.

[67] Mark Stoodley, Kennth Ma, and Marius Lut. Real-time java, part 2: Comparing compilation techniques [online]. 2007. Available from: `http://www.ibm.com/developerworks/java/library/j-rtj2/index.html`.

[68] Dan Sugalski. Implementing an interpreter [online]. Available from: `http://www.sidhe.org/%7Edan/presentations/Parrot%20Implementation.ppt`. Notes for slide 21.

[69] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journals, Java Performance Issue*, 39(1), Feb. 2000.

[70] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28(1):134–174, 2006.

[71] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proc. of the Workshop on Interpreters, Virtual Machines and Emulators*, 2003.

[72] V. Sundaresan, D. Maier, P Ramarao, and M Stoodley. Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler. In *Proc. of the 4th Intl. Symp. on Code Generation and Optimization*, pages 87–97, Mar. 2006.

[73] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, message, and performance: how they coexist in Self. *IEEE-COMPUTER*, 25(10):53–64, Oct. 1992.

[74] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[75] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and operand specialization for Tcl VM performance. In *Proc. 2nd IVME*, pages 42–50, 2004.

[76] Benjamin Vitale and Mathew Zaleski. Alternative dispatch techniques for the Tcl vm interpreter. In *Proceeedings of Tcl'2005: The 12th Annual Tcl/Tk Conference*, October 2005. Available from: `http://www.cs.toronto.edu/syslab/pubs/tcl2005-vitale-zaleski.pdf`.

[77] John Whaley. Partial method compilation using dynamic profile information. In *Proc. of the 16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–179, Oct. 2001.

[78] Wikipedia. Ucsd p-system — wikipedia, the free encyclopedia, 2007. [Online; accessed 15-May-2007]. Available from: `http://en.wikipedia.org/w/index.php?title=UCSD_p-System&oldid=117632578%`.

[79] Tom Wilkinson. The Kaffe java virtual machine [online]. Available from: `http://www.kaffe.org/`.

[80] Mathew Zaleski, Marc Berndl, and Angela Demke Brown. Mixed mode execution with context threading. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005.