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

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* (HLL 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 HLL 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 HLL 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 which are easier to compile than than the inlined method bodies compiled by current systems. In addition, hot traces 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 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.

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 compiled 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. "big bang"

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, 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. In case the cold code does eventually run, the generated code and the runtime that supports it must deal with the complexities of late binding [69]. 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 a simple arithmetic operation 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 which 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 [25].

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.

todo: allude to interp traces

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 instructions, 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, a Java virtual machine, and `ocamlrun`, an Ocaml interpreter [12], 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 run time 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 started 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 [49], with a trace oriented JIT compiler. We built Yeti, (graduallY Extensible Trace Interpreter) in five stages: First, we repackaged all virtual instruction bodies as callable. Our initial implementation executed only single virtual instructions which were dispatched from a simple dispatch loop. Second, we identified *linear blocks*, or sequences of virtual instructions ending in

branches. Third, we extended our system to identify and dispatch *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 our 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 inter-procedural 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

todo: touch upon branch prediction

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 a selective inlining interpreter, previously the state of the art.

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 conclusions 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 [61]. 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 [75, 15] and Java bytecode [48] 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[2]. Smalltalk [33], Self [70] 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 complex machine and ABI dependent sequence of 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 [64].

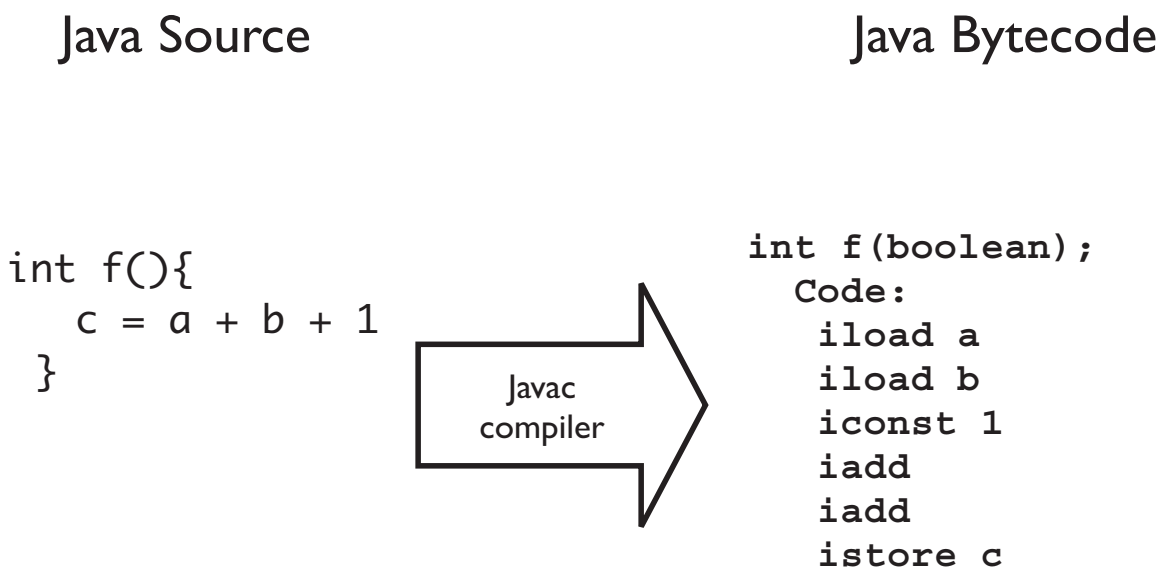


Figure 2.1: Example Java Virtual Program showing source (on the left) and Java virtual instructions, or bytecodes, on the right.

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 part of a 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 which are passed on a run time stack. For instance, in Figure 2.1, the `iadd` instruction pops the top two slots of the run time 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 operand stack. *Local variable array* slots, or *lva* slots, are used to store local variables and parameters. The simple function shown needs only four local variable slots (referred to as lva[0] through lva[3] in the figure). 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 *operand stack* is used to maintain the expression stack used for all calculations and parameter passing. In general "load" form bytecodes push values in lva slots onto the operand stack. Bytecodes with "store" in their mnemonic typically pop the value on top of the operand 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

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

interpreter. For instance, designers may prefer a representation that word-aligns all immediate parameters regardless of their size. This would be less compact, but faster to access, than the original byte code on most architectures.

todo: move bodies here?

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. 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 operand 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 showed that on modern processors branch mispredictions caused by dispatch branches are a serious drain on performance [25, 26].

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 issues, 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 for the HP3000 [44] as early as 1979. Deutsch and Schiffman [21] built an early Just in Time (JIT) compiler for Smalltalk that obtained a speedup of about two relative to 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[16][3]. More recent JIT compilers, for instance have increased the performance further[1, 3, 66]. 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 [63, slide 28].

[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 operand stack, one register-to-register arithmetic instruction to do the real work and a store (push) back to the new top of the operand stack.

[3]These benchmarks are singled out because they eventually were adopted by the SPEC consortium to be part of the SPECjvm98 [62] benchmark suite.

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 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 supports explicit, built-in support for threads. *Thread synchronization* refers mostly to the functionality that allows one one thread at a time to access certain regions of code. 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.

```
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

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 runtime by a combination of the method signature and the class of the invoked upon object. Thus callsites are said to be *polymorphic* as the invoked upon object may turn out to be one of potentially many classes.

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 runtime 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

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

`toString` method on each element as in Figure 2.2.

There are literally hundreds of definitions of `toString` in a Java system and in most 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. 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 inter-procedural optimizations and thus results in relatively slow systems. This is discussed below.

Impact of Polymorphism on Optimization

Inter-procedural 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 inter-procedural optimization as carried out by an optimizing C or FORTRAN compiler is simply not possible[51].

In the absence of inter-procedural 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 inter-procedural 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 sep-

arated 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, inter-procedural 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 symbol before they start to 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 a 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 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 [69].

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 pay backs 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 [47, 4, 31, 35, 34].

Although the semi-automatic systems did not enable dramatic performance improvements across the board, this may be a consequence of the performance baseline they compared themselves to. 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 which 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

[5] This roughly describes how JamVM and SableVM, and perhaps other interpreters handle late binding.

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 [45]. 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 onto a display [53]. 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 specific blit. 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 [47] built FABIUS, a dynamic optimization system for the research language ML [31]. 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 [4], Grant [35, 34] 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 run-time constant. Later systems allowed the user to identify only the program variables that are run-time 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 run-time 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 [35, 34]. 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 run-time 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 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 [22].

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 runtime. 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*, as invented by Deutsch and Schiffman [21] 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[39] 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 much less than for effectively polymorphic ones.

2.4.2 Smalltalk and Self

Smalltalk, an early object oriented language, 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 [32].

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.

[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.

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 [13, 40, 39, 41] 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. in [70]. 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 known) and static inlining (to expose the guarded code to inter-procedural 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 [13] 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 [70]. 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

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

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 [13] 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 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:

[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.

> 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.[1, 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 defines 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 [52].

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 [40]. 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) [28] which 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 run time 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 [74].

Suganuma et al. [67] 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. 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 were 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 [6, 23, 5] is a same-ISA binary optimizer. Dynamo initially interprets a binary executable program, detecting 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 [11], Mojo [14], Transmeta's CMS [20], 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. Recently, Hiniker et al. [37] described improvements to NET that reduce replication and handle loops better.

Software trace caches are efficient structures for dynamic optimization. Bruening and Duesterwald [8] 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 improve the security of legacy applications [11, 10].

One interesting application of DynamoRIO was by Sullivan et al [68]. 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 have described as the context problem (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 [7] reports similar performance results running a similarly modified Kaffe JVM [76].

Last Executed Iteration (LEI)

Hiniker, Hazelwood and Smith performed a simulation study evaluating enhancements to the basic Dynamo trace selection heuristics [37]. 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.

2.6 Hotpath

Gal, Probst and Franz describe the Hotpath project. Hotpath extends JamVM (one of the interpreters we use for our experiments) to be a trace oriented mixed-mode system [30]. 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) [19] 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 benchmarks like LU, SOR and Linpack, 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" [30, page 151]. The optimization techniques they describe seem complimentary to the overall architecture we propose in Chapter 6.

2.7 Summary? Tail? What to call this section?

todo: fix section head

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. In this chapter we described

how HP Dynamo defines a simple but effective heuristic that can be used to find hot paths of a running program and also how to link on new paths as they become hot. 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 current dispatch techniques. In Chapter 2 we defined dispatch as the mechanism used by a high level 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 [27, 65].

Section 3.1 describes switch dispatch, the simplest dispatch technique. Section 3.2 introduces call threading, which figures prominently in our techniques. 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. Finally, Section 3.6 describes various related work that improves or eliminates dispatch overhead.

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 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 expression is executed. 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 corresponding the 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.

todo: add detail to bodies

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.

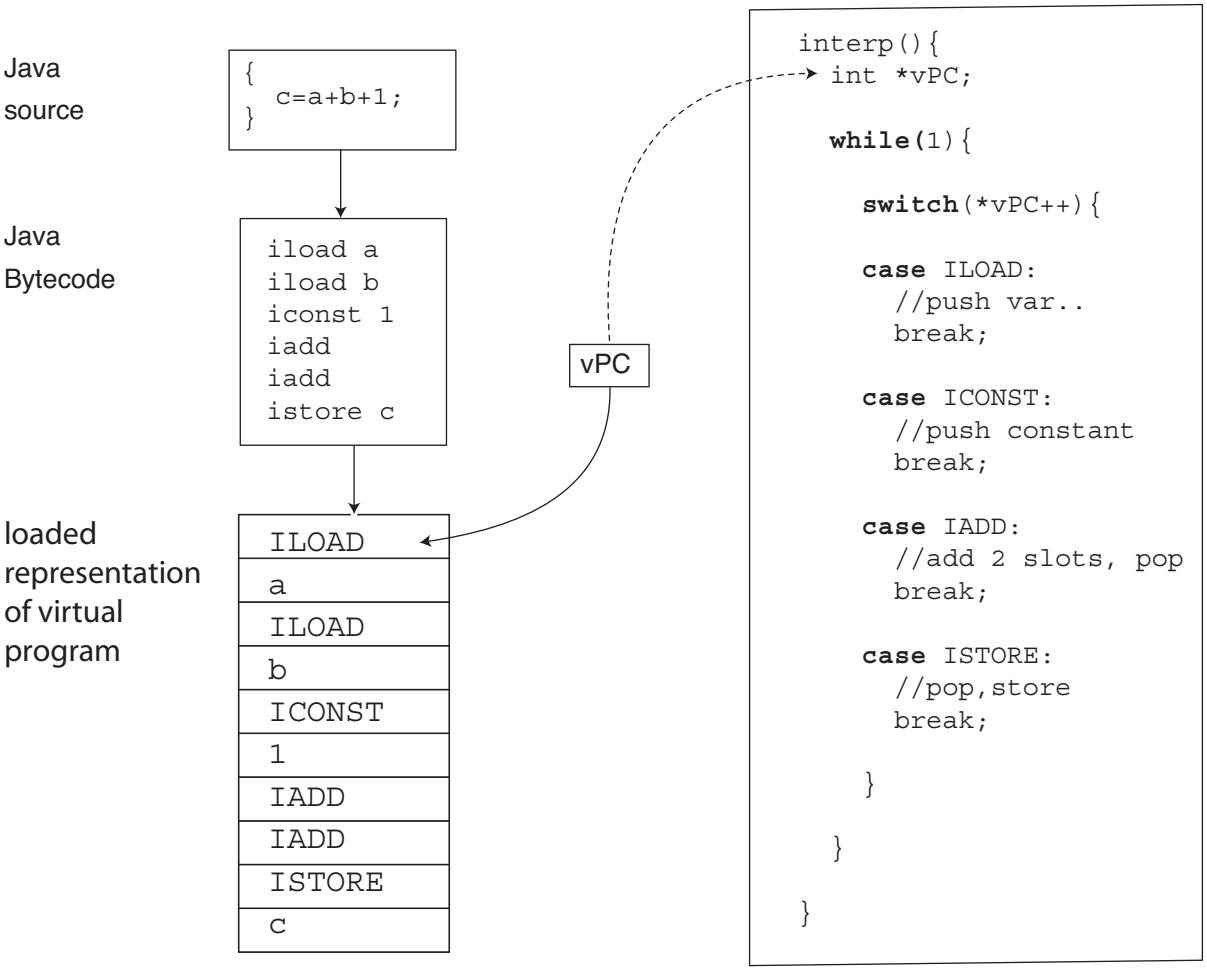


Figure 3.1: A switch interpreter loads each virtual instruction as a virtual opcode, or token, corresponding to the case of the switch statement that implements it.

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 [24]. 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 to 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.

todo: fig doesn't work for Angela

3.3 Direct Threading

As shown on the left of Figure 3.3, a virtual program is loaded into a direct-threaded interpreter by constructing a *list of addresses*, one for each virtual instruction in the program, pointing to the body for that instruction. We refer to this list as the *Direct Threading Table*, or DTT, and refer to locations in the DTT as *slots*. Virtual instruction operands are also stored in the DTT, immediately after the address of the corresponding body.

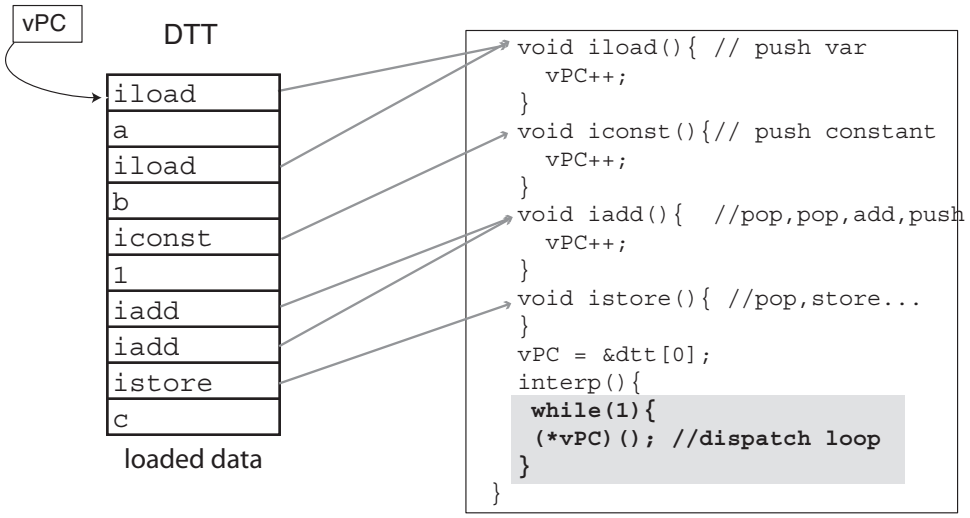


Figure 3.2: A direct call threaded interpreter packages each virtual instruction body as a function. The shaded box highlights the dispatch loop showing how instructions are called through a function pointer. Direct call threading requires the loaded representation of the program to indicate the *address*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

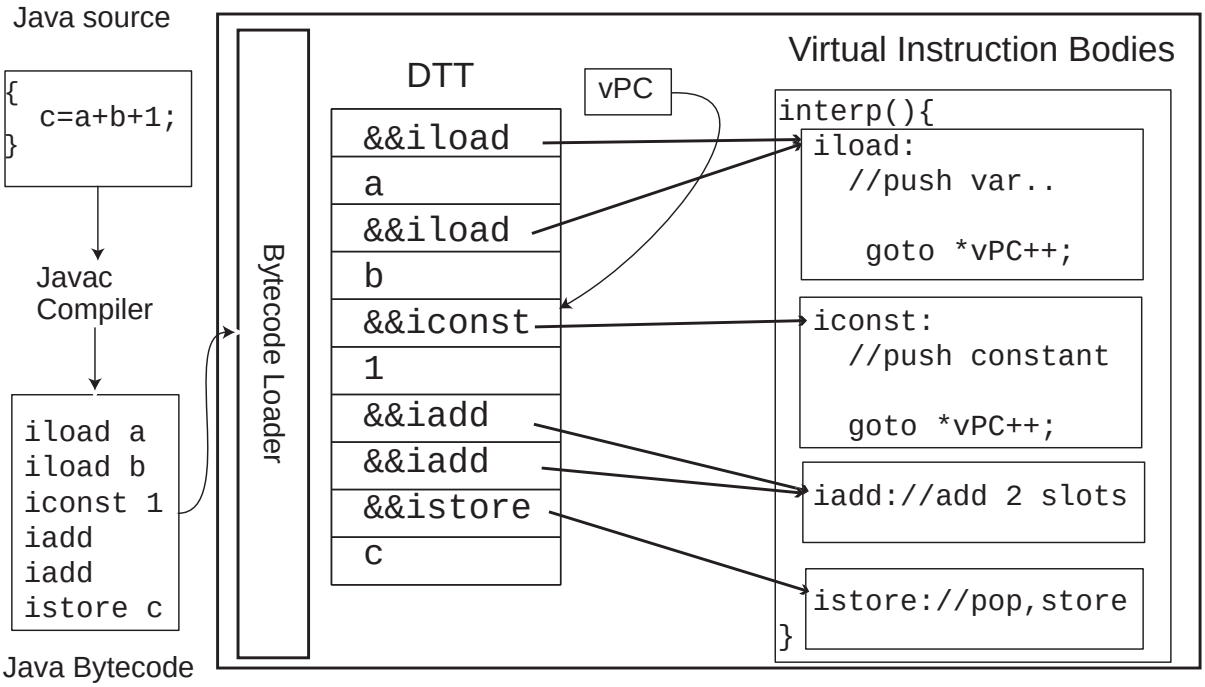


Figure 3.3: Direct Threaded Interpreter showing how Java Source code compiled to Java bytecode is loaded into the Direct Threading Table (DTT). The virtual instruction bodies are written in a single C function, each identified by a separate label. The double-ampersand (&&) shown in the DTT is gcc syntax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

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

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 second branch to its contents.

Interpretation begins by initializing the `vPC` to the first slot in the DTT, and then jumping to the address stored there. Each body then ends by transferring control to the next instruction, shown in Figure 3.3 as `goto *vPC++`. 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 address and computed goto extensions, making direct threading quite portable.

Direct threading requires fewer instructions and is faster than 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

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 [36]. 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 [38]. (There is no mention of a BTB in the PowerPC 970 programmers manual [42].) Direct threading confounds the BTB because all instances of a given virtual instruction compete for the same BTB slot. The performance impact of this can be hard to predict. For instance, if a tight loop of the virtual program happens

to contain a sequence of unique virtual instructions then the BTB may successfully predict each one. On the other hand, if the sequence contains duplicate virtual instructions, like the pair of `iload` instructions in Figures 3.3 and 2.1, the BTB may mispredict all of them.

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 [50]. 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. 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* [38] whereas the PPC970 uses a similar structure called the *link stack* [42].

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 is usually wrong for direct threaded interpreters [25, 26]. 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.

To the hardware 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 `vPC`. We refer to this lack of correlation between the hardware `pc` and `vPC` as the *context problem*.

3.6 Optimizing Dispatch

Much of the work on interpreters has focused on how to optimize dispatch. Kogge [46] remains a definitive description of many threaded code dispatch techniques. These 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, discussed next. Later, we will discuss superinstructions and replication, which are in the second class. We are particularly interested in subroutine threading and replication because they both provide context to the branch prediction hardware.

Some Forth interpreters use subroutine-threaded dispatch. Here, a loaded virtual program is not represented as a list of body addresses, but instead as a sequence of native *call*s to the bodies, which end with native *return*s. Curley [18, 17] 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 [58] 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 [17]. On modern hardware the cost of the *call* and *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 demonstrate this effect on a few modern CPUs.

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 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 [27]).

Piumarta [56] presents *selective inlining*. It constructs superinstructions when the virtual program is loaded. They are created in a relatively portable way, by `memcpy`'ing the native 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 any relative branches remain in the body. Typically this excludes bodies making C function calls. This technique was first documented earlier [60], 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 5.5, our technique is separate from, but supports and indeed facilitates, inlining optimizations.

Languages, like Java, that require run-time 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 point to a string naming the callee. The argument will be re-written, to point to a descriptor of the now resolved callee, the first time the virtual instruction executes. 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 [29].

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* [26] 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 [72].

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 [26]. Replication implemented by inlining opcode bodies reduces the number of dispatches, and therefore, the average dispatch overhead [56]. In the extreme, one could create a copy for each instruction, eliminating misprediction entirely. This technique results in significant code growth, which may [72] or may not [26] cause cache misses.

In summary, misprediction of the indirect branches used by a direct threaded interpreter to dispatch virtual instructions limits its performance on modern CPUs because of the context problem. 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.

3.7 Eloquent Linkage to Next Chapter

In this chapter we have described how a few common interpretation techniques work. In the next chapter we will describe a new technique for interpretation that deals with the context problem as we defined it in Section 3.5 above. Our technique, context threading, performs well compared to the interpretation techniques we have described in this chapter. In Chapter 6, we show how an enhanced version of a direct call threaded interpreter can be extended to be a trace based dynamic compiler.

Chapter 4

Design and Implementation of Efficient Interpretation

Over and above the desirable qualitative properties introduced in Chapter 1, an interpreter organized around callable bodies can outperform a direct threaded interpreter. In this chapter we describe techniques that exploit the performance potential of callable bodies.

Recently, we realized that the implementation of return branch predictor stacks by modern microprocessor CPU implementations has a great impact on the performance of subroutine threading. Subroutine threading, described in Section 3.6, packages virtual instructions as lightweight callable routines. The return branch predictor stack in a modern CPU should perfectly predict the return. We realized that if we generated a sequence of direct call instructions at load time, one for each virtual instruction, then the calls would pose no prediction problems either.

Our implementation combines the dispatch of Forth, subroutine threading, with the internal representation of direct threading, the direct threading table (or DTT), to achieve a dispatch technique that eliminates branch mispredictions for straight-line code. Our implementation of subroutine threading requires the generation of only one type of machine instruction, a direct call, so although it is not portable its hardware dependency is isolated to a few lines of code.

By generating increasingly complicated code at load time we extended ourtechnique to eliminate branch mispredictions caused by dispatching virtual branch instructions as well. In the next chapter , where we evaluate the performance of subroutine threading, we will show that even the simplest subroutine threading technique runs the SPECjvm98 suite about 20% faster than direct threading on PPC970 and Pentium 4 processors.

Subroutine threading minimally affects code size. Although direct-threaded interpreters are known to have poor branch prediction properties they are also known to have a small instruction cache footprint [59]. 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. This is in contrast to the current state of the art as described in Section 3.6. There, we described various techniques for improving branch prediction by replicating entire bodies. The effect of these techniques is to trade instruction cache size for better branch prediction.

4.1 Understanding Branches

Before motivating our design, we first note two main issues. 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, sequential (virtual) execution is just as expensive as handling control transfers, 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

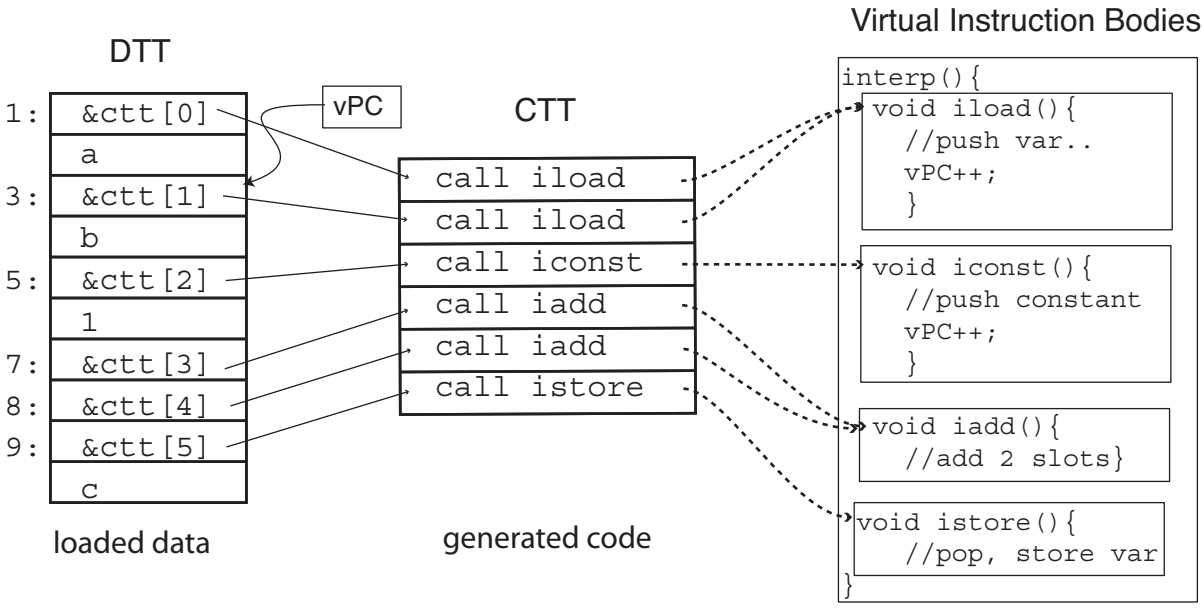


Figure 4.1: Subroutine Threaded Interpreter showing how the CTT contains one generated direct call instruction for each virtual instruction and how the first entry in the DTT corresponding to each virtual instruction points to generated code to dispatch it. Callable bodies are shown here as nested functions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

that exist in the virtual program are effectively hidden from the microprocessor.

The fundamental goal 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, we exploit the different types of hardware prediction resources to handle the different types of virtual control flow transfers. 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. We strive to maintain the property that the virtual program counter is precisely correlated with the physical program counter and in fact, with when all our techniques are combined there is a one-to-one mapping between them at most control flow points.

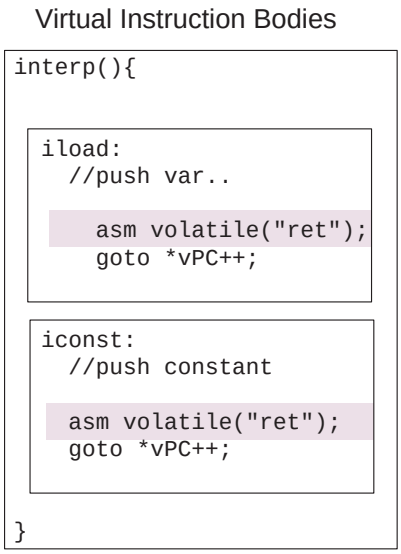


Figure 4.2: Direct threaded bodies retrofitted as callable routines by inserting inline assembler return instructions. This example is for Pentium 4 and hence ends each body with a `ret` instruction. The `asm volatile` statement is an extension to the C language , inline assembler, provided by gcc and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

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. The increase in code size caused by aggressive inlining, however, has the potential to overwhelm the benefits with the cost of increased instruction cache misses [72].

Rather than eliminate dispatch, we propose an alternative organization for the interpreter in which native call and return instructions are used. Conceptually, this approach is elegant because subroutines are a natural unit of abstraction to express the implementations of virtual instructions.

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. We use the term *Context Threading*, because the hardware address of each call instruction in the CTT provides execution context to the hardware, most importantly, to the branch predictors.

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 Direct Threading Table (DTT) is still necessary to store immediate operands, and to correctly resolve virtual control transfer instructions. In direct threading, entries in the DTT point to opcode 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. It is not obvious whether a call and return is more or less expensive to execute than an indirect jump, but that is not the issue. Although the cost of subroutine threading is two control transfers, versus one for direct threading, this cost is outweighed by the benefit of eliminating a large source of unpredictable branches.

4.3 Handling Virtual Branches

Subroutine threading handles the branches that implement the dispatch of straight-line virtual instructions; however, the actual 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, one relating to shared indirect branch prediction resources, and one 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`, marked (a) in the figure. Even for this simple virtual branch, prediction is problematic, because *all* instances of `ifeq` instructions in the virtual program share a single indirect branch instruction (and hence have a single prediction context). A simple solution is to generate replicas of the indirect branch instruction in the CTT immediately

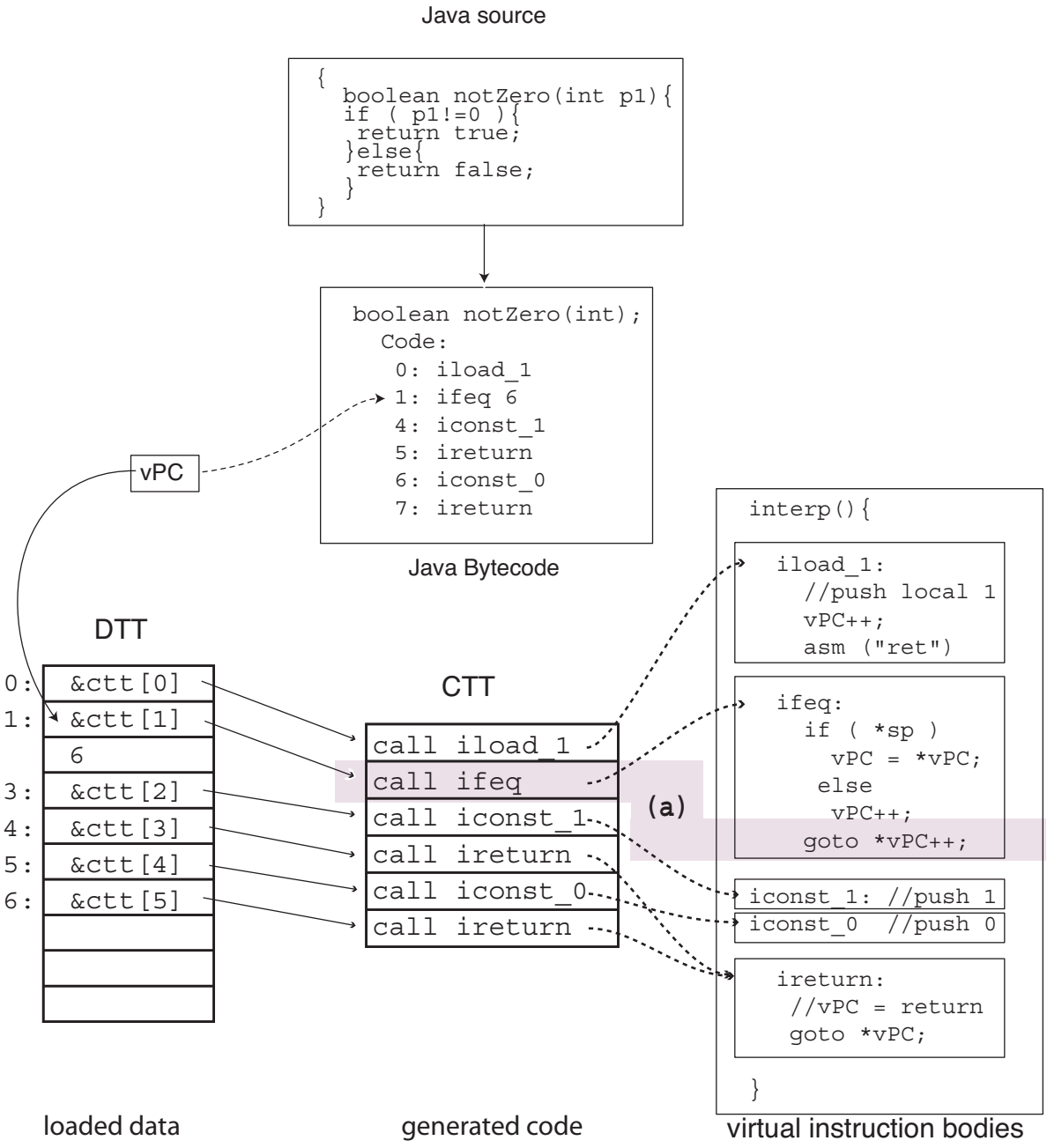


Figure 4.3: Subroutine Threading does not not address branch instructions. Unlike straight line virtual instructions virtual branch bodies end with an indirect branch destination (just like direct threading).

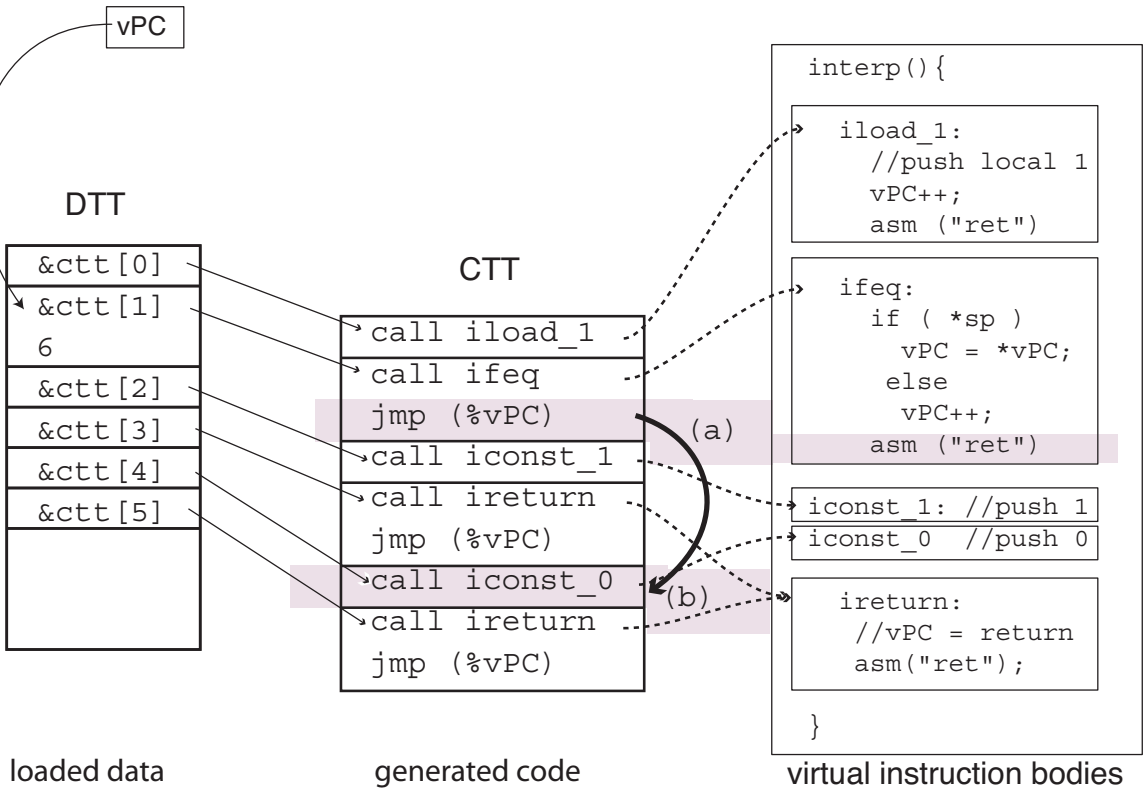


Figure 4.4: Context threading with branch replication illustrating the "replicated" indirect branch (a) in the CTT. The fact that the indirect branch corresponds to only one virtual instruction gives it better prediction context. The heavy arrow from (a) to (b) is followed when the virtual branch is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

following the call to the branching opcode body. Branching opcode bodies now end with native return, which return to the CTT before executing the replicated indirect branch. As a consequence, each virtual branch instruction now has its own hardware context. We refer to this technique as *branch replication*. Figure 4.4 illustrates how branch replication works.

Branch replication is attractive because it is simple and produces the desired context with a minimum of replicated 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 hard-

ware. Due to these limitations, we only use branch replication for indirect virtual branches and exceptions[1].

For all other branches we generate code for the bodies of virtual branch instructions into the CTT. We refer to this as *branch inlining*. In the process of inlining, we convert indirect branches into direct branches, where possible. On the Pentium this reduces pressure on the branch taken 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 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, a couple of i-cache lines, of additional space. Figure 4.5 shows an example of inlining the `ifeq` branch instruction. The machine code, shaded in the figure, implements the same if-then-else logic as the original direct threaded virtual instruction body. In the figure we assume key interpreter variables like the virtual PC and expression stack pointer exist in dedicated registers. This is the technique we use in Ocaml on both the Pentium 4 and the PowerPC, and SableVM on the PowerPC, but not for SableVM on the Pentium, where they are stored in stack slots instead. We show Intel instructions in the figure but similar code must be generated on the PowerPC. The inlined conditional branch instruction (`jne`, marked (a) in the figure) is fully exposed to the Pentium's conditional branch prediction hardware.

An obvious challenge with branch inlining is that the generated code is not portable and assumes detailed knowledge of the virtual bodies it must interoperate with. For instance, in Figure 4.5 the generated code must know that the Pentium's `esi` register has been dedicated to the `vPC`.

[1] Ocaml defines explicit exception virtual instructions

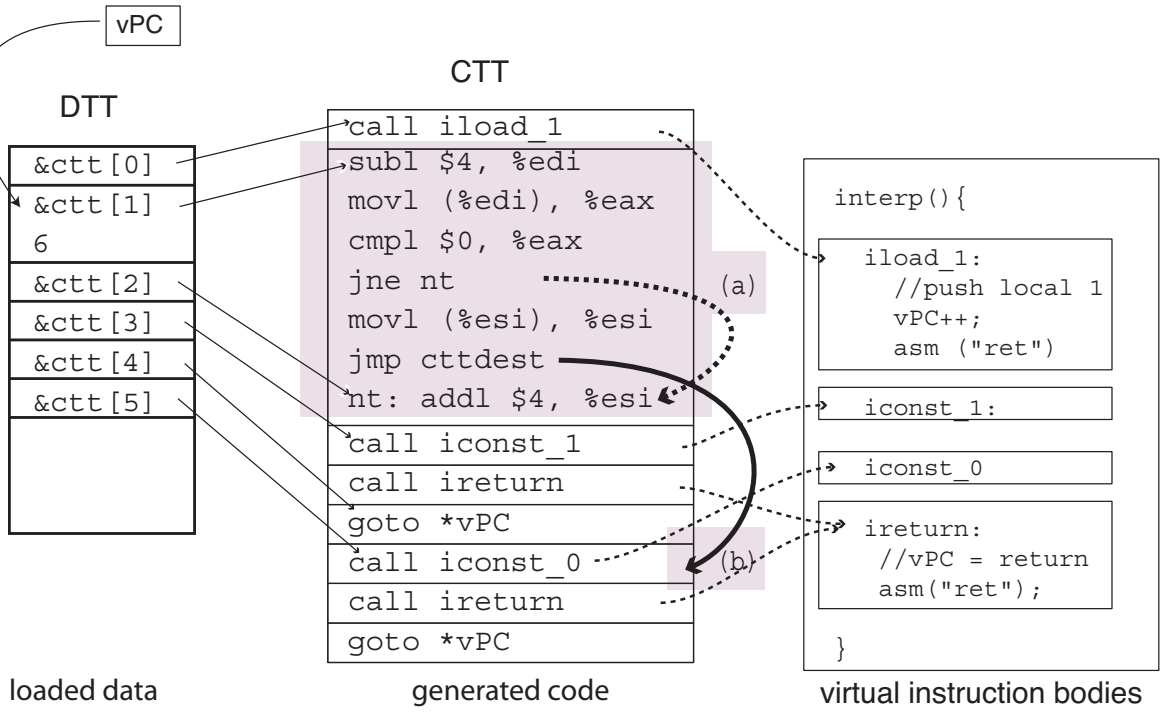


Figure 4.5: Context Threaded VM Interpreter: Branch Inlining on Pentium. The generated code (shaded) assumes the `vPC` is in register `esi` and the Java expression stack pointer is in register `edi`. The dashed arrow (a) illustrates the inlined conditional branch instruction, now fully exposed to the branch prediction hardware, and the heavy arrow (b) illustrates a direct branch implementing the not taken path.

4.4 Handling Virtual Call and Return

The only significant source of control transfers that remain in the virtual program are 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 potentially has many destinations, one for each callsite of the method. As noted previously, the hardware already has an elegant solution to this problem for native code 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 call body should transfer control to the start of the callee. We begin at a virtual call instruction (see label "(a)" in the figure). The method invocation body, Java's `invokestatic` in the figure, creates a new frame for the callee, etc, and then sets the `vPC` to the entry point of the callee and executes

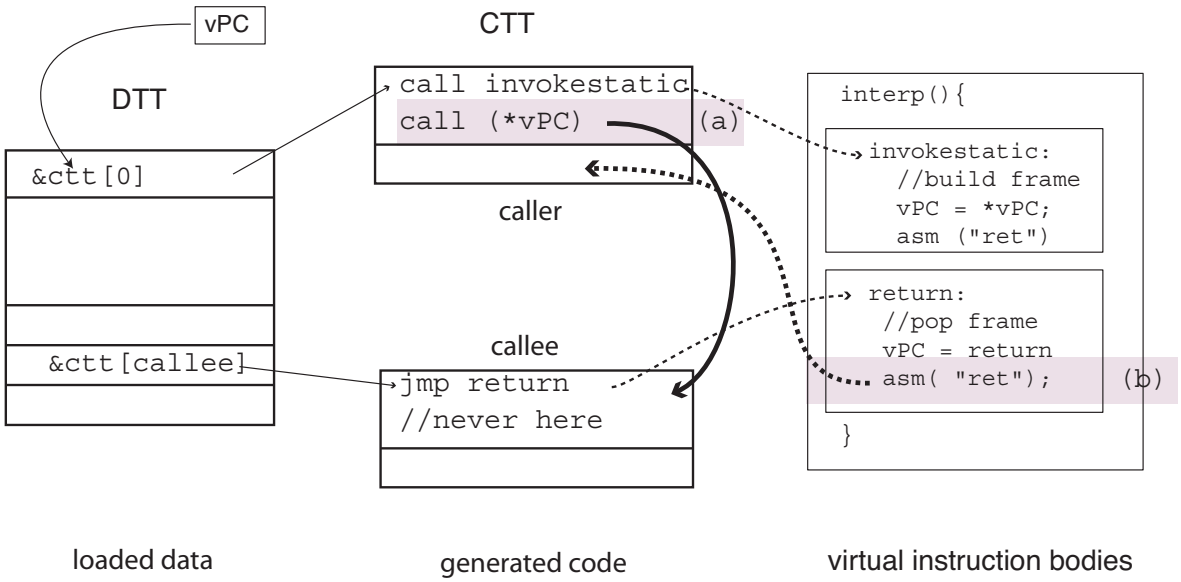


Figure 4.6: Context Threading Apply-Return Inlining on Pentium. The generated code *calls* `invokestatic` but *jumps* (instruction at (b) is a `jmp`) to the return .

a native *return*, an x86 `ret` in the figure, back to the CTT. Similar to branch replication, we insert a new native *call indirect* instruction following "(a)" in the CTT to transfer control to the start of the callee (solid arrow from "(a)" to "(b)" in the figure). The call indirect causes the next location in the CTT to be pushed 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, we emit a native direct *branch* , an x86 `jmp` in the figure, to dispatch the body of the virtual return (before label "(b)".) The direct branch avoids perturbing the return address stack. We modify the body of the virtual return to end with a native *return* instruction, which now transfers control all the way back to the instruction following the original virtual call (dotted arrow from "(b)" to "(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

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

hardware stack against the virtual program stack. For instance, in OCaml, exceptions unwind the virtual machine stack; the hardware stack must be unwound in a corresponding manner. Second, some run-time environments are extremely sensitive to hardware stack manipulations, since they use or modify the machine stack pointer for their own purposes (such as handling signals). In such cases, it is possible to create a separate stack structure and swap between the two at virtual call and return points. This approach would introduce significant overhead, and is only justified if apply/return inlining provides a substantial performance benefit.

4.5 Eloquent Linkage to Next Chapter

Having described our design and its general implementation, we now evaluate its effectiveness on real interpreters.

Chapter 5

Evaluation of Context Threading

In this section, we evaluate our interpretation technique by comparing its performance to direct threading and direct-threaded selective inlining. Our complete technique, context threading, is actually a combination of subroutine threading, branch inlining and apply/return inlining. We evaluate the contribution of each of these techniques to the overall impact using two virtual machines and three microprocessor architectures.

We begin by describing our experimental setup in Section 5.1. We then investigate how effectively our techniques address pipeline branch hazards in Section 5.4.1, and the overall effect on execution time in Section 5.4.2. Finally, Section 5.5 demonstrates that context threading is complementary to inlining resulting in performance comparable to SableVM's implementation of selective inlining.

5.1 Virtual Machines, Benchmarks and Platforms

We evaluated our techniques by modifying interpreters for Java and Ocaml to run on Pentium IV, PowerPC 7410 and PPC970.

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

Table 5.2: Description of SpecJVM 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

5.1.1 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.

5.1.2 SableVM

SableVM is a Java Virtual Machine built for quick interpretation, implementing lazy method loading and a novel bi-directional virtual function lookup table. Hardware signals are used to handle exceptions. Most importantly for our purposes, SableVM implements multiple dispatch mechanisms, including switch, direct threading, and selective inlining (which SableVM calls *inline threading* [29]). The support for multiple dispatch mechanisms facilitated our work to add context threading.

5.1.3 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 may 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 contribute to the behavior.

5.1.4 SableVM Benchmarks

SableVM experiments were run on the complete SPECjvm98 [62] suite (`compress`, `db`, `mpegaudio`, `raytrace`, `mtrt`, `jack`, `jess` and `javac`), one large object oriented application (`soot` [71]) and one scientific application (`scimark` [57]). Table 5.2 summarizes the key characteristics of these benchmarks.

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

5.2 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 show how effectively our techniques improve 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].

5.3 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 [50]. 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.

[2]MPT events are counted with performance counter 8 by setting the P4 CCCR to 0x0003b000 and the ESCR to value 0xc001004 [43]

Table 5.3: (a) Guide to Technique description.

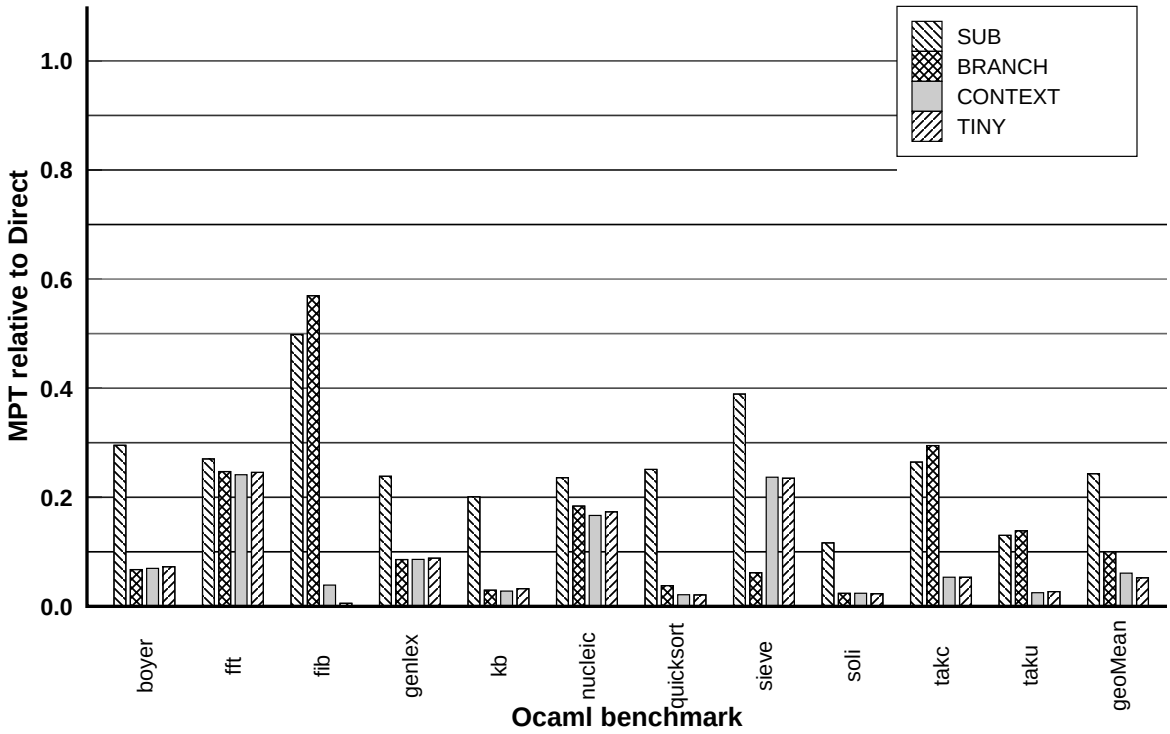
Technique	Key	Description
Subroutine Threading	SUB	Section 4.2
Branch Inlining	BRANCH	Section 4.3
Context Threading	CONTEXT	Section 4.4
Tiny Inlining	TINY	Section 5.5
Selective Inlining (sablevm)	SELECT	Section 3.6

(b) Guide to performance data figures.

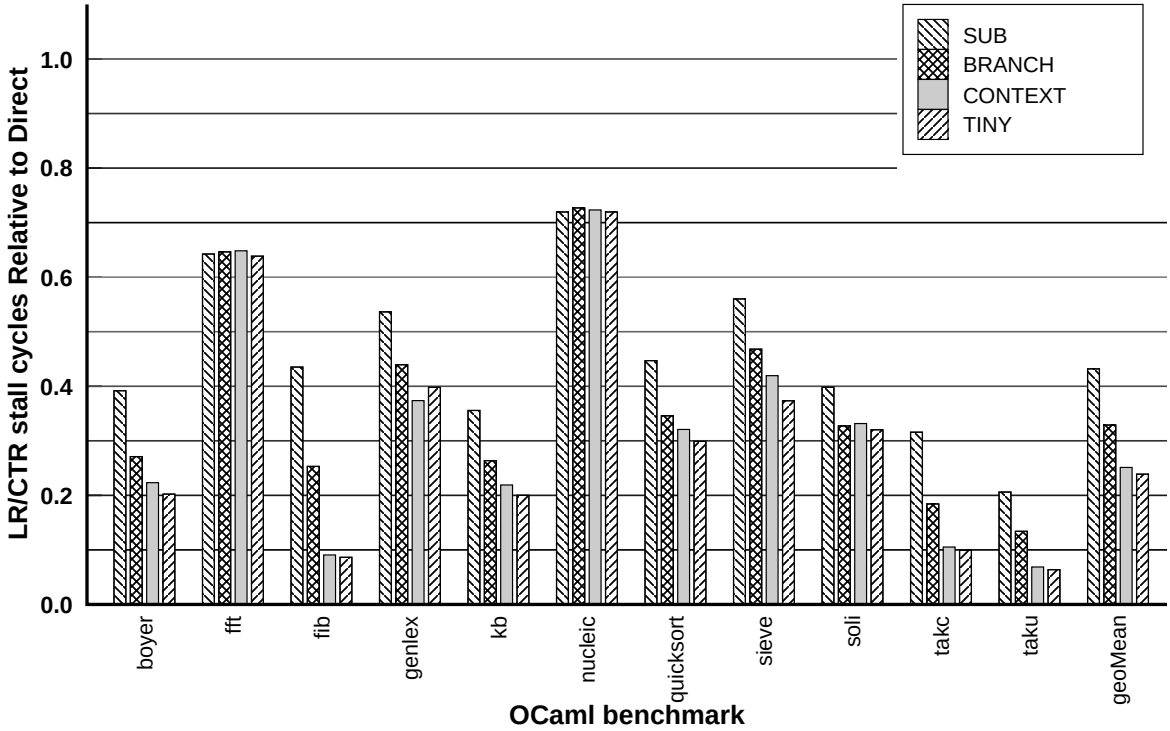
Interpreter	Hazards	P4/PPC7410 Performance	PPC970 time
Ocaml	Figure 5.1 on the following page	Figure 5.3 on page 68	Figure 5.5 on page 70 (a)
Java (SableVM)	Figure 5.2 on page 65	Figure 5.4 on page 69	Figure 5.5 on page 70 (b)

5.4 Interpreting the data

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, presumably because it performs the best.) We give the absolute execution times and branch hazard statistics for each benchmark and platform using direct threading in Tables 5.1 and 5.2. 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 BRANCH); and our complete context threading implementation which includes apply/return inlining (labeled CONTEXT. We include bars for selective inlining in SableVM (labeled SELECT) and our own simple inlining technique (labeled TINY) to facilitate comparisons, although inlining results are not discussed until Section 5.5. We do not show a bar for direct threading because it would, by definition, have height 1.0. See Table 5.3

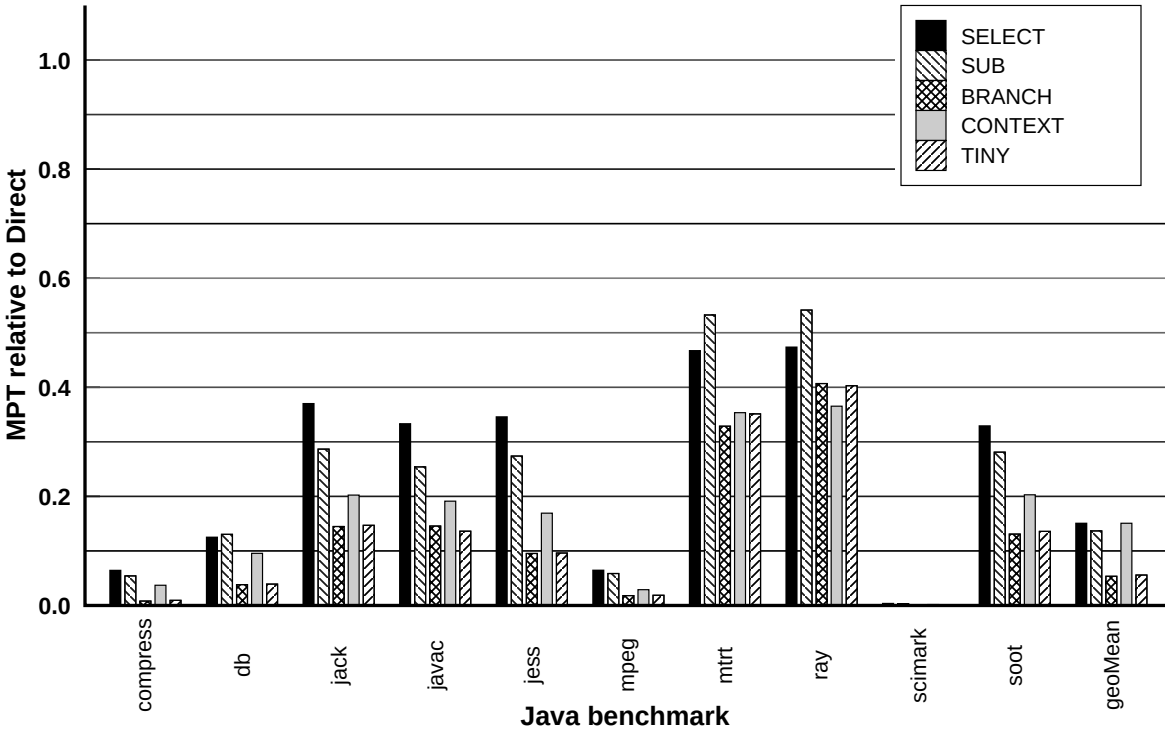


(a) Pentium 4 Mispredicted Taken Branches

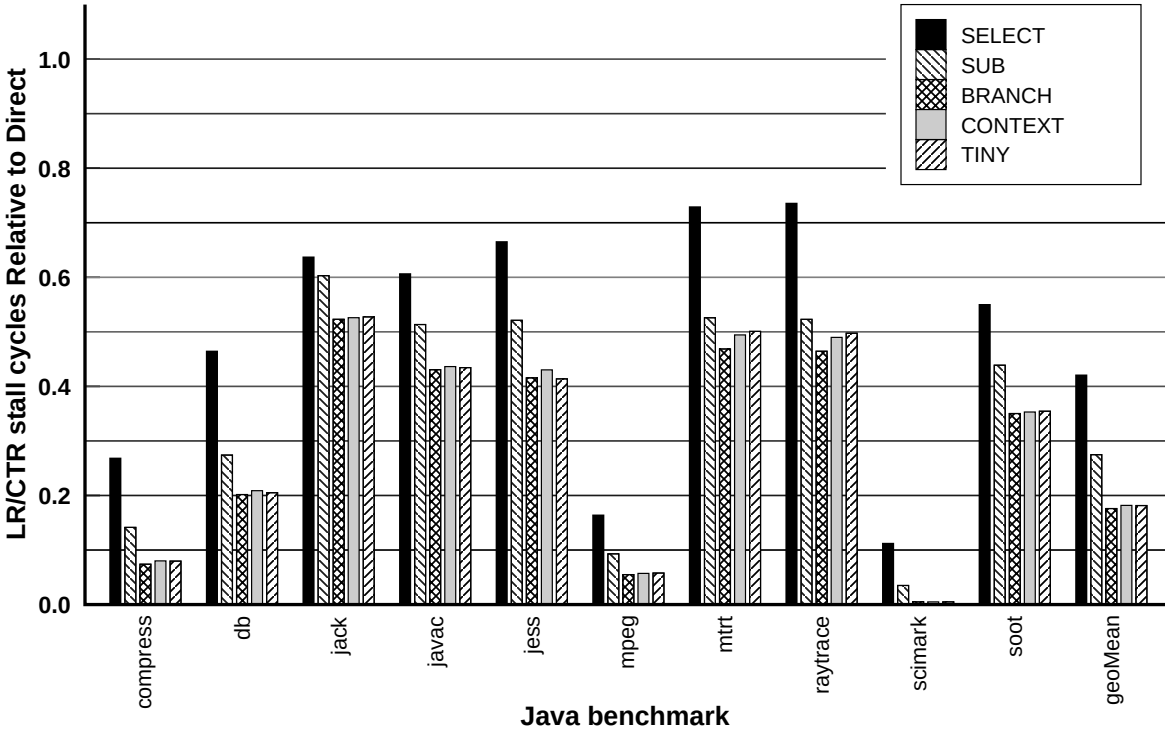


(b) PPC 7410 lr-ctr stall cycles

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

5.4.1 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 each Figure, 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 all straight-line bytecodes. Branch inlining has the next largest effect, again as expected, 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 branch inlining and so the BTB predicts adequately. Hence, inlining the conditional branches cannot help.

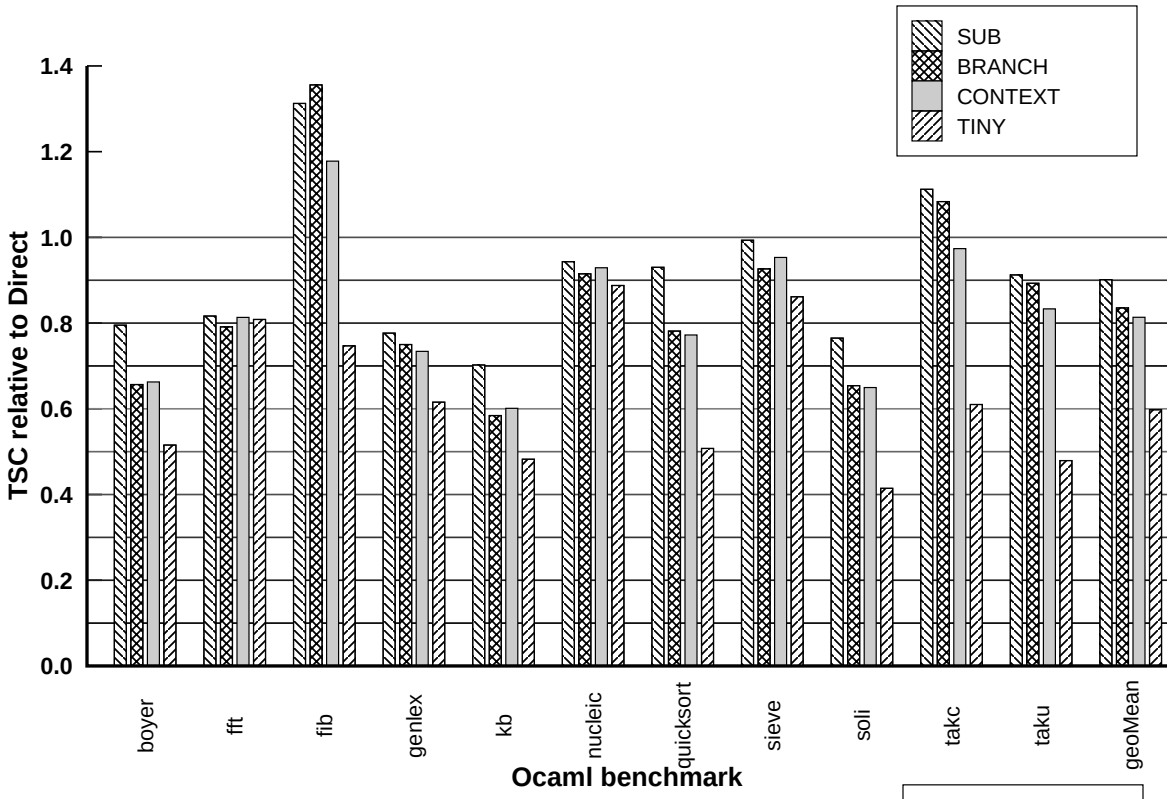
Interestingly, the same three OCaml micro benchmarks `Fib`, `takc` and `taku` that challenge branch inlining on the P4 also reap the greatest benefit from apply/return inlining, as shown in Figure 5.1(a). Due to the recursive nature of these benchmarks, their performance is dominated by the behavior of virtual calls and returns. Thus, we expect predicting the returns to have significant impact. However, the effect does not hold for all recursive benchmarks. For `sieve`, on the P4, the result of apply/return inlining is an increase in MPT, while for the non-recursive OCaml benchmarks, the overall effect on both platforms is a small improvement.

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 and reduces the effectiveness of the return address stack predictor, as can be seen in the bar labeled CONTEXT in Figure 5.2(a). On the PPC7410, the effect of apply/return inlining on LR/CTR stalls is very small for SableVM.

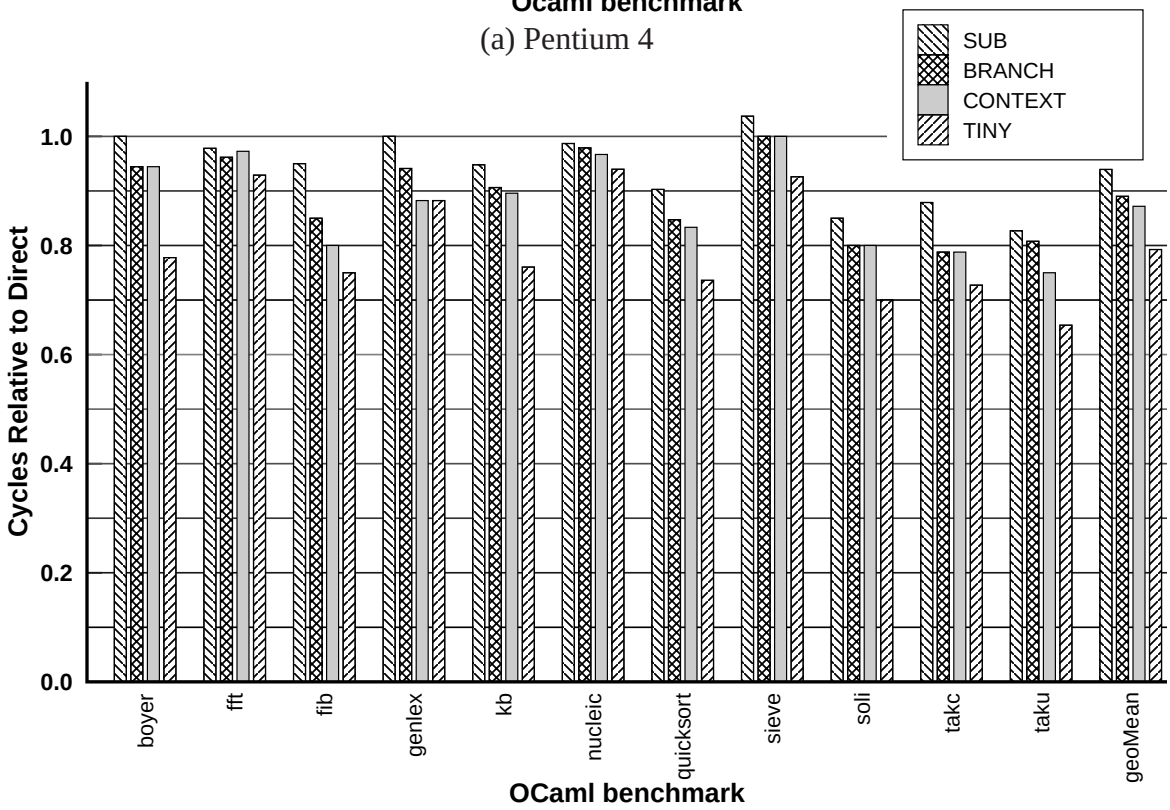
Having shown that our techniques can significantly reduce pipeline branch hazards, we now examine the impact of these reductions on overall execution time.

5.4.2 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.

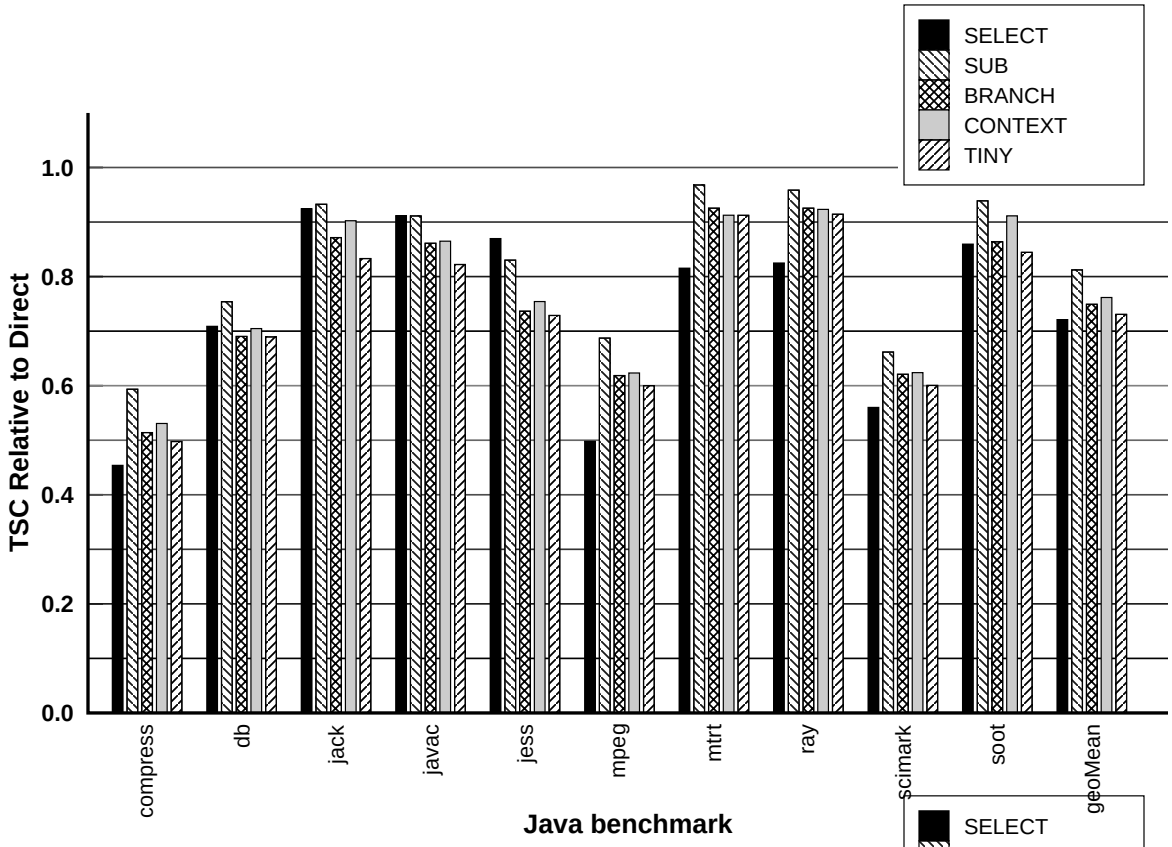


(a) Pentium 4

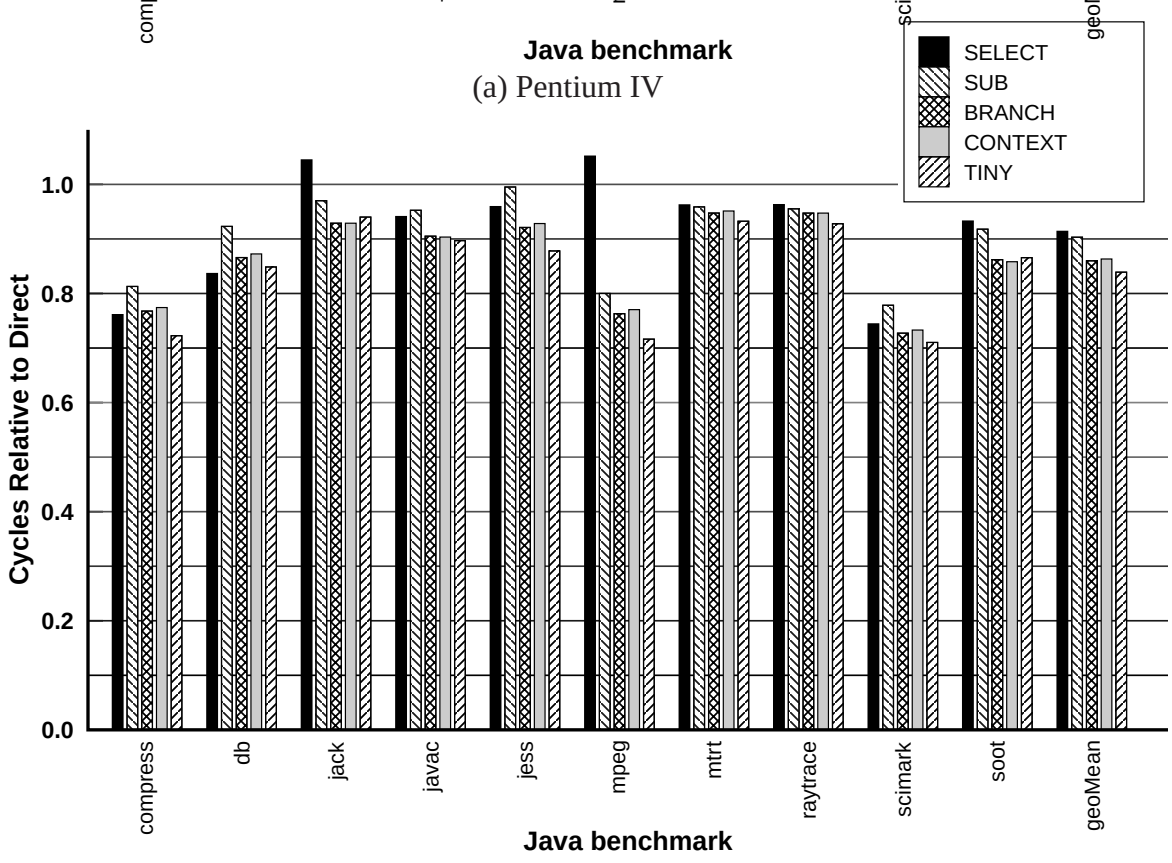


(b) PPC7410

Figure 5.3: OCaml Elapsed Time Relative to Direct Threading

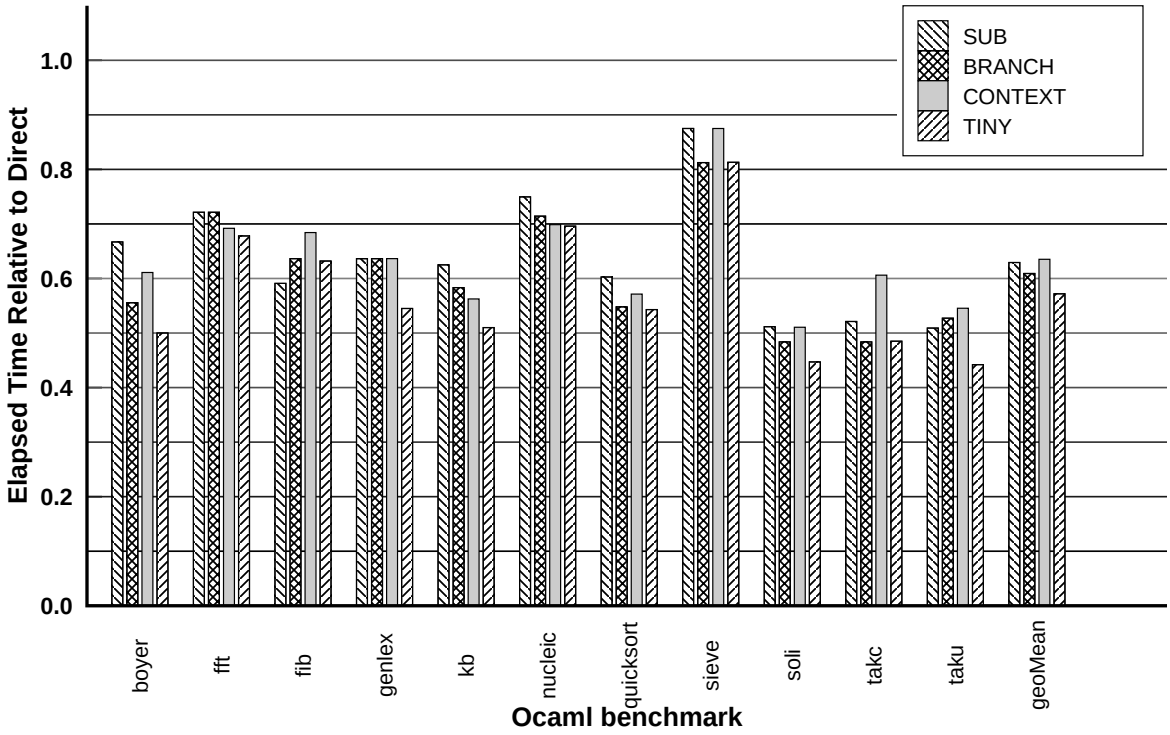


(a) Pentium IV

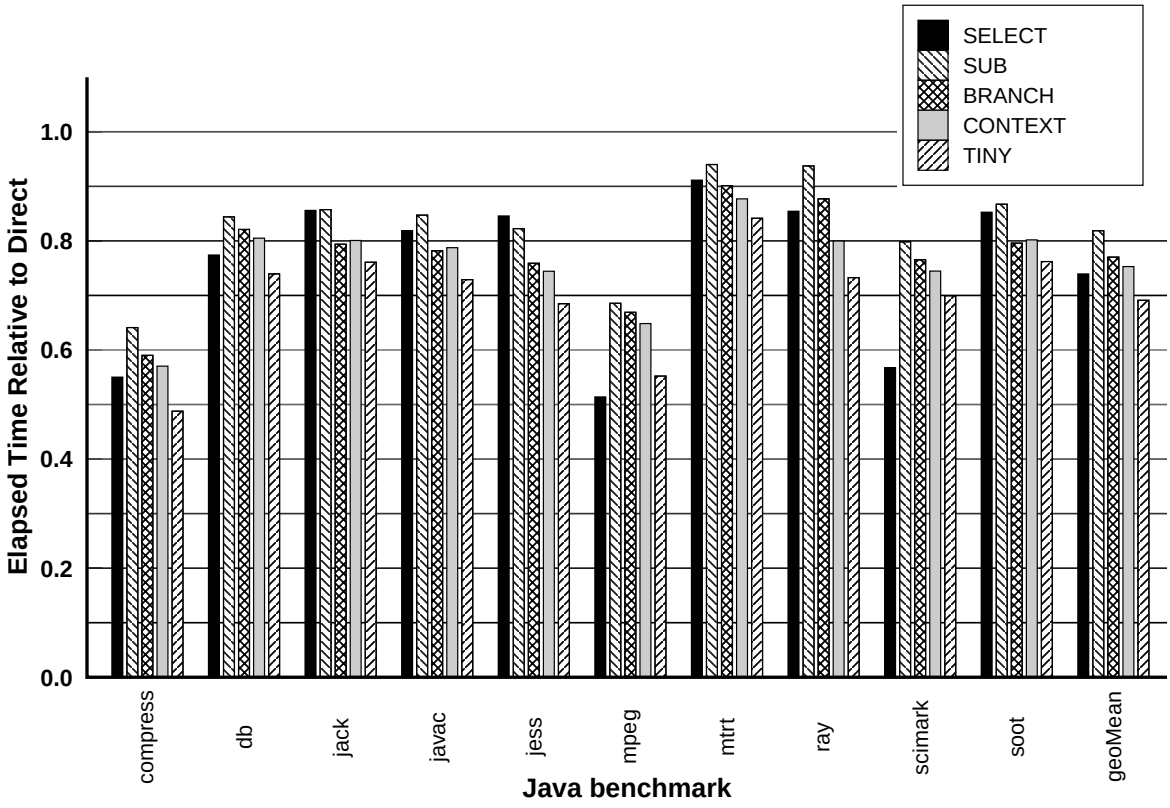


(b) PPC 7410

Figure 5.4: SableVM Elapsed Time Relative to Direct Threading



(a) OCaml PPC970 elapsed (real) seconds



(b) SableVM PPC970 elapsed (real) seconds

Figure 5.5: PPC970 Elapsed Time 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 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, context threading, 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 so few instances of a few virtual instructions 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 tech-

niques.

5.5 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 black bar labeled SELECT shows our measurements of Gagnon's selective inlining implementation for SableVM [29]. 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 inlining strategies. 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 minimal. In

Table 5.4: Detailed comparison of selective inlining vs the combination of context+tiny (SableVM). Numbers are elapsed time relative to direct threading for SableVM. $\triangle(S-C)$ is the the difference between selective inlining and context threading. $\triangle(S-T)$ is the difference between selective inlining and the combination of context threading and tiny inlining.

Arch	Context (C)	Selective (S)	Tiny (T)	Δ (S-C)	Δ (S-T)
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

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 sophisticated selective inlining implementation. On PowerPC, we outperform SableVM by 7.8% for the PPC7410 and 4.8% for the PPC970.

The main performance issue with direct-threaded interpretation is pipeline branch hazards caused by the context problem. Context threading solves this problem by correctly deploying branch prediction resources, and as a result, outperforms direct threading by a wide margin. Once the pipelines are full, the cost of executing dispatch instructions is significant. A suitable technique for addressing this overhead is inlining, and we have shown that context threading is compatible with inlining using our "tiny" heuristic. With this simple approach, context threading achieves performance roughly equivalent to, and occasionally better than, selective inlining.

5.6 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 how a Tcl interpreter typically executes an order of magnitude more cycles per dispatched virtual

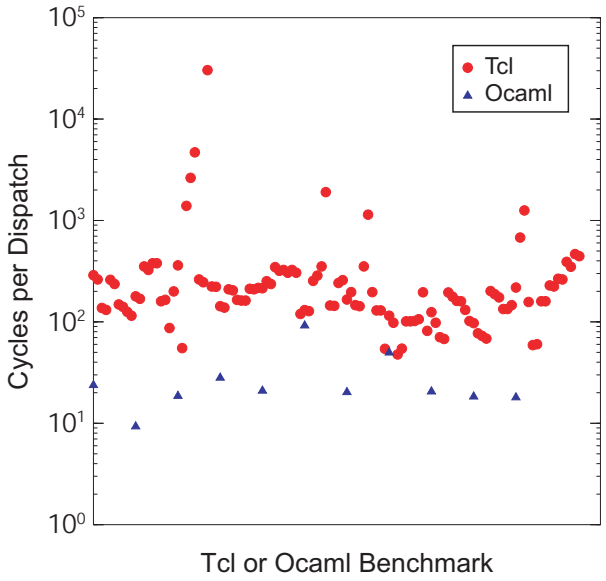


Figure 5.6: Reproduction of [73, Figure 1] showing cycles run per virtual instructions dispatched for various Tcl and Ocaml benchmarks .

instruction than Ocaml. Another perspective is that Ocaml executes 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. In fact, the geometric mean of 500 Tcl benchmarks speeds up only 5.4 % on a UltraSPARC III. As shown in Figure 5.7 subroutine threading alone improved our Ocaml benchmark much more.

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 suggests that for programs that execute only a small number of virtual instructions the load time overhead of generating code in the CTT is an issue.

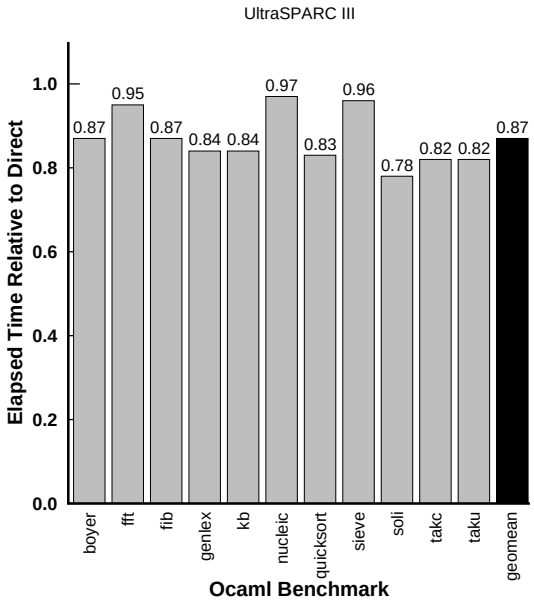


Figure 5.7: Elapsed time of subroutine threading relative to direct threading for Ocaml on UltraSPARC III.

5.7 Eloquent Linkage to Next Chapter

Our experimentation with subroutine threading has established that calling virtual instruction bodies is an efficient way of implementing straight-line regions of a virtual program. Branch inlining is an effective way of eliminating the branch mispredictions caused by virtual branches. Inlining bodies into the CTT is simple option that can increase the performance of context threading to the state of the art of interpretation techniques.

These results also contain some warnings. First, our attempt 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 (and much more labor intensive to implement) branch inlining that we improved the performance of virtual branches significantly. Second, the slowdown observed amongst a few TCL benchmarks (that dispatched very few virtual instructions) raise the concern that even the load time overhead of subroutine threading may be too high.

These results inform our design of a gradually extensible interpreter, presented in Chapter

6, in a few ways. First, linear regions of the program should be interpreted using subroutine threading. Second, the conditional branch instructions used to implement virtual branches should be inlined so that they are exposed to the hardware's conditional branch predictors. Third, loading should be lazy so that the, potentially large, regions of the program that never run do not incur any load time overhead.

Chapter 6

Design and Implementation of YETI

Early on we realized that organizing virtual bodies as lightweight routines would make it possible to call them from generated code and that this has potential to simplify bringing up a JIT. At the same time, we realized that we could expand our use of the DTT to dispatch execution units of any size, including basic blocks and traces, and that this would allow us to gradually extend our system to more ambitious execution units. We knew that it was necessary to interpose instrumentation between the virtual instructions but we could not see a simple way of doing it. We went ahead regardless and built an instrumentation infrastructure centered around code generation. The general idea was to initially generate trampolines, which we called interposers, that would call instrumentation before and after the dispatch of each virtual instruction. The infrastructure was very efficient (probably more efficient than the system we will describe in this chapter) but quite difficult to debug. We extended our system until it could identify basic blocks and traces [77]. Its main drawback was that a lot of work was required to build a profiling system that ran no faster than direct threading. This, we felt, was not "gradual" enough. Fortunately, a better idea came to mind.

Instead of loading the program as described for context threading, Yeti runs a program by initially dispatching single virtual instruction bodies from an instrumented dispatch loop reminiscent of direct call threading. Instrumentation added to the dispatch loop detects execution

units, initially basic blocks, then traces, then linked traces. As execution units are generated their address is installed into the DTT. Consequently the system speeds up as more time is spent in execution units and less time on dispatch.

6.1 Instrumentation

In Yeti, as in subroutine threading, the vPC points into the DTT where each virtual instruction is represented as one or more contiguous slots. The loaded representation of the program has been elaborated significantly – now the first DTT slot of each instruction points to an instance of a *dispatcher* structure. The dispatcher structure contains four key fields. The execution unit to be dispatched (initially a virtual instruction body, hence the name) is stored in the *body* field. The *preworker* and *postworker* fields store the addresses of the instrumentation routines to be called before and after the dispatch of the execution unit. Finally, the dispatcher has a *payload* field, which is a chunk of profiling or other data that the instrumentation needs to associate with an execution unit. Payload structures are used to describe virtual instructions, basic blocks, or traces.

Despite being slow, a dispatch loop is very attractive because it makes it easy to instrument the execution of a virtual program. Figure 6.1 shows how instrumentation can be interposed before and after the dispatch of each virtual instruction. The figure illustrates a generic form of dispatch loop (the shaded rectangle in the lower right) where the actual instrumentation routines to be called are implemented as function pointers accessible via the vPC . In addition we pass a payload to each instrumentation call. The disadvantage of this approach is that the dispatch of the instrumentation is burdened by the overhead of a call through a function pointer. This is not a problem because Yeti actually deploys several specialized dispatch loops and the generic version illustrated in Figure 6.1 only executes a small proportion of the time.

Our strategy for identifying regions of a virtual program requires every thread to execute in one of several execution “modes”. For instance, when generating a trace, a thread will be in

trace generation mode. Each thread has associated with it a *thread context structure* (tcs) which includes various mode bits as well as the *history list*, which is used to accumulate regions of the virtual program.

6.2 Loading

When a method is first loaded we don't know which parts of it will be executed. As each instruction is loaded it is initialized to a shared dispatcher structure. There is one shared dispatcher for each kind of virtual instruction. One instance is shared for all `iload` instructions, another instance for all `iadd` instructions, and so on. Thus, minimal work is done at load time for instructions that never run. On the other hand, a shared dispatcher cannot be used to profile instructions that do execute. Hence, the shared dispatcher is replaced by a new, non-shared, instance of a *block discovery dispatcher* when the postworker of the shared dispatcher runs for the first time. The job of the block discovery dispatcher is to identify new basic blocks.

6.3 Basic Block Detection

When the preworker of a block discovery dispatcher executes for the first time, and the thread is *not* currently recording a region, the program is about to enter a basic block that has never run before. When this occurs we switch the thread into *block recording mode* by setting a bit in the thread context structure. Figure 6.1 illustrates the discovery of the basic block of our running example. The postworker called following the execution of each instruction has appended the instruction's payload to the thread's history list. When a branch instruction is encountered by a thread in block recording mode, the end of the current basic block has been reached, so the history list is used to generate an execution unit for the basic block. Figure 6.2 illustrates the situation just after the collection of the basic block has finished. The dispatcher at the entry point of the basic block has been replaced by a new *basic block dispatcher* with a new payload

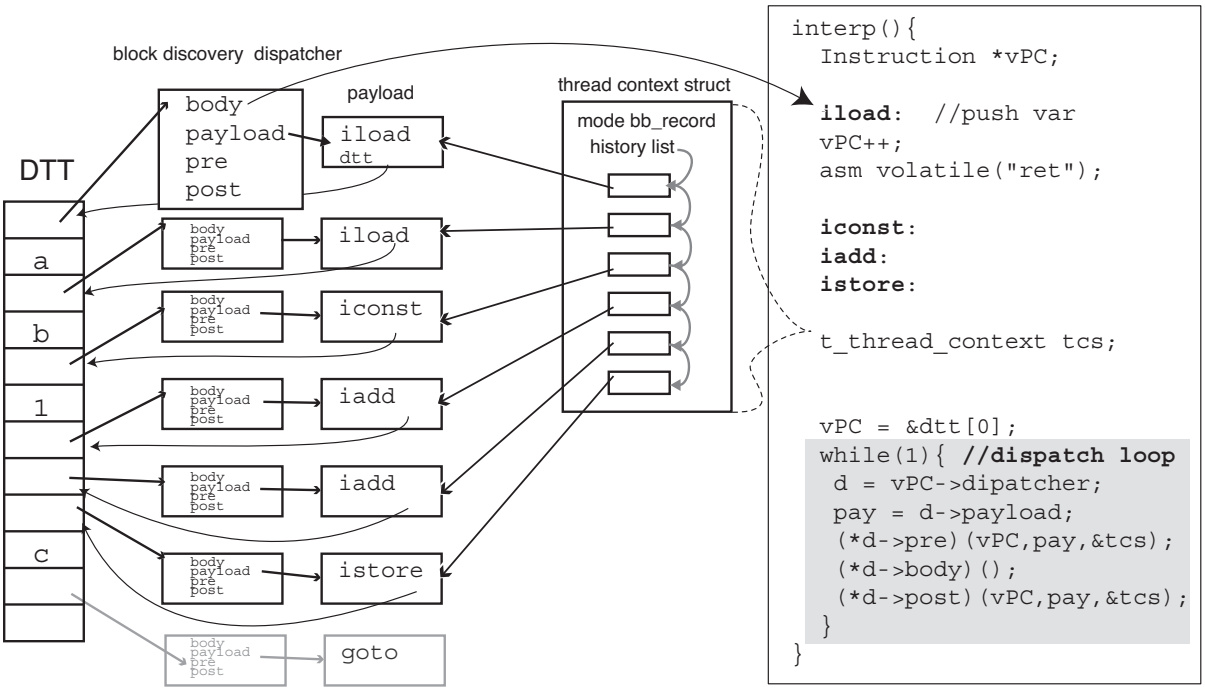


Figure 6.1: Shows a region of the DTT during block recording mode. The body of each block discovery dispatcher points to the corresponding virtual instruction body (Only the body for the first iload is shown). The dispatcher's payload field points to instances of instruction payload. The thread context struct is shown as tcs.

created from the history list. The body field of the basic block dispatcher points to a subroutine threading style execution unit that has been generated for the basic block. The job of the basic block dispatcher will be to search for traces.

6.4 Trace Selection

The postworker of a basic block dispatcher is called after the last virtual instruction of the block has been dispatched. Since basic blocks end with branches, after executing the last instruction the vPC points to one of the successors of the basic block. If the vPC of the destination is *less* than the vPC of the virtual branch instruction, 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 it enters *trace recording mode*. In

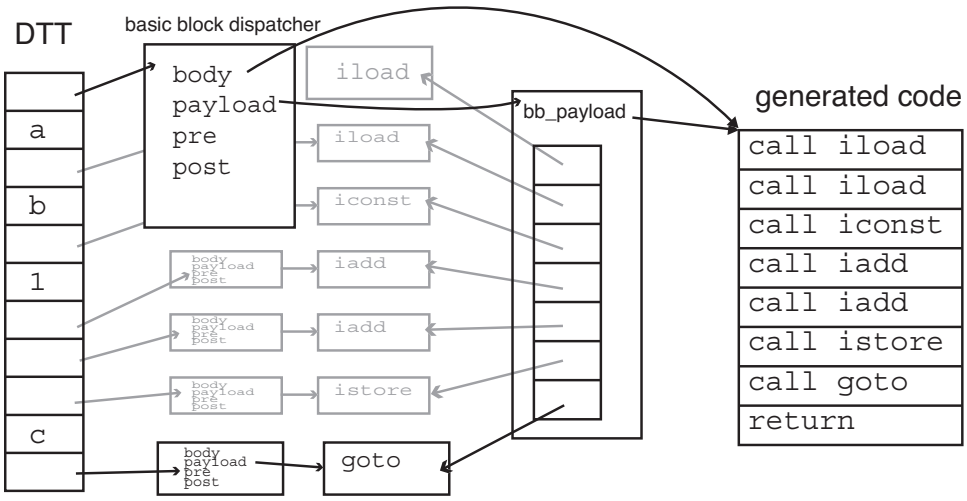


Figure 6.2: Shows a region of the DTT just after block recording mode has finished.

trace recording mode, much like in basic block recording mode, the postworker adds each basic block to a history list. The situation is very similar to that illustrated in Figure 6.1, except the history list describes basic blocks. Our system, like Dynamo, ends a trace (i) when it reaches a reverse branch, (ii) when it finds a cycle, or (iii) when it contains too many (currently 100) basic blocks. When trace generation ends, a new *trace dispatcher* is created and installed. This is quite similar to Figure 6.2 except that a trace dispatcher is installed and the generated code is complicated by the need to support trace exits. The payload of a trace dispatcher includes a table of *trace exit descriptors,* one for each basic block in the trace. 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. Trace training mode uses a specialized dispatch loop that calls instrumentation before and after dispatching each virtual instruction in the trace. In principle, almost any detail of the virtual machine's state could be recorded. Currently, we record the class of every Java object upon which a virtual method is invoked. When training is complete, code is generated for the trace as illustrated by Figure 6.3. Before we discuss code generation, we need to describe the runtime of the trace system and especially the operation of trace exits.

6.5 Trace Exit Runtime

The runtime of traces is complicated by the need to support trace exits, which 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 than during trace generation. Generated guard 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 in the thread's context structure and then returns to the dispatch loop, which immediately calls the postworker corresponding to the trace. The postworker determines which trace exit occurred by examining the thread context structure. Conceptually, the postworker has only a few things it can do:

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

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 new trace.

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 tail of the trace exit handler to jump directly to the destination trace instead of returning to the dispatch loop. Subsequently execution stays in the trace 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.

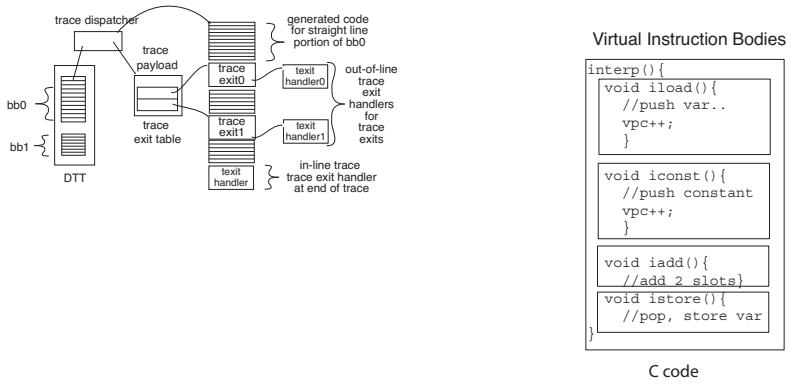


Figure 6.3: Schematic of a trace

The first challenge we essentially punt on. 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 trace cache memory.

The second challenge concerns the efficient selection of a destination trace to which to link, and the mechanics used to branch there. To choose a destination, we follow the heuristic developed by Dynamo for regular branches – that is, we link to destinations in the order they are encountered. 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 we know the specific values the vPC could have, we can hard-wire the comparand in the generated code. In fact, we can generate a sequence of compares checking for two or more destinations. 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 a PIC, used to dispatch polymorphic methods, as discussed in Section **??**.

6.6 Generating code for traces

Generating a trace is made up of two main tasks, generating a trace exit handler for each trace exit and generating the main body of the trace. Trace generation starts with the list of basic blocks that were selected. We will use these to access the virtual instructions making up the trace. After a few training runs we have also have fine-grained profiling information on the precise values that occur during the execution of the trace. These values will be used to devirtualize selected virtual method invocations.

6.6.1 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 code that performs a compare followed by a conditional branch. PowerPC code for this case appears in Figure 6.4. The sense of the conditional branch is adjusted so that the branch is always not-taken for the on-trace path. More complex virtual branch instructions, and especially those with multiple destinations, are handled differently. Instead of generating inlined code for the branch we generate a call to the virtual branch body instead. This will have the side effect of setting the vPC to the destination of the branch. Since only one destination can be on-trace, and since we know the exact vPC value corresponding to it, we then generate a compare immediate of the vPC to the hardwired constant value of the on-trace destination. Following the compare we generate a conditional branch to the corresponding trace exit handler. The result is that execution leaves the trace if the vPC set by the dispatched body was different from the vPC observed during trace generation. Polymorphic method dispatch is handled this way if it cannot be optimized as described in Section 6.6.3.

Trace exit handlers have three further roles not mentioned so far. First, since traces may contain compiled code, it may be necessary to flush values held in registers back to the Java expression stack before returning to regular interpretation. Code is generated to do this in each

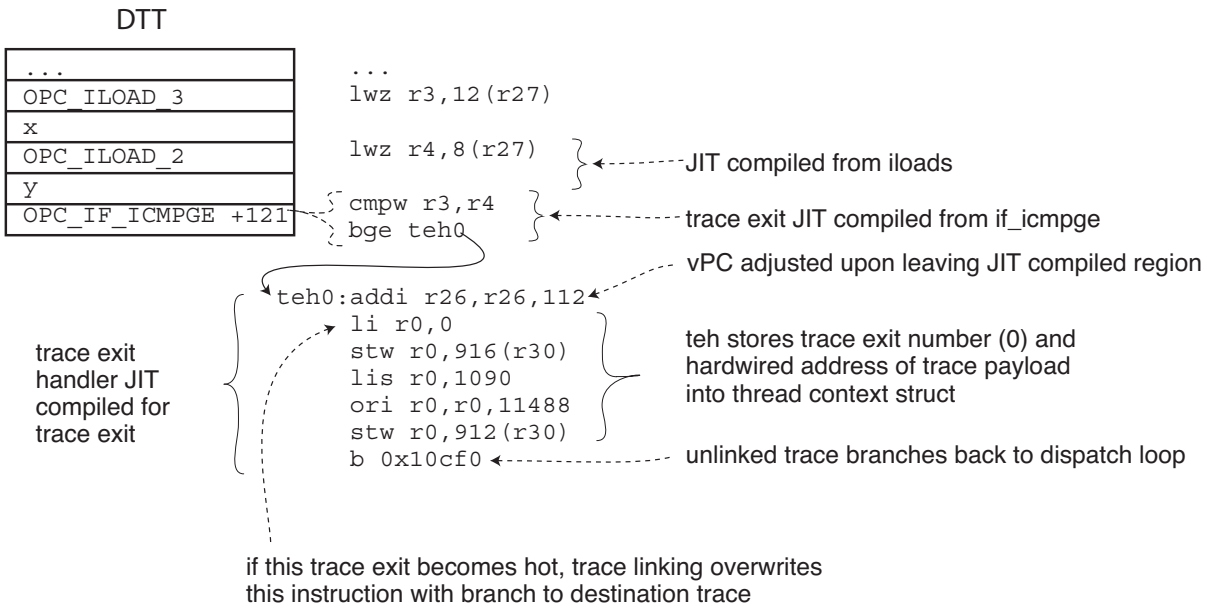


Figure 6.4: PowerPC code for a trace exit and trace exit handler. The generated code assumes that the `vPC` has been assigned `r26`, base of the local variables `r27` and the Java method frame pointer `r30`.

trace exit handler. Second, some interpreter state may have to be updated. For instance, in Figure 6.4, the trace exit handler adjusts the `vPC`. Third, 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.

6.6.2 Code Generation

The body of a trace is made up of straight-line sections of code, corresponding to the body of each basic block, interspersed with trace exits generated from the virtual branches ending each basic block. The JIT therefore has three types of information to start with. First, there is a list of virtual instructions making up each basic block in the trace. Enough information is cached in the trace payload to determine the virtual opcode and virtual address of each instruction in the trace. Second, there is a trace exit corresponding to the branch ending each basic block. The trace exit stores information like the `vPC` of the off-trace destination of the trace. Third,

there may be profiling information that was cached when the trace ran in training mode.

At this phase of our research we have not invested any effort in generating optimized code for the straight-line portions of a trace. Instead, we implemented a simple one pass JIT compiler. The goals of our JIT are modest. First, it should perform a similar function as branch inlining (Section 4.3) to ensure that code generated for trace exits exposes the conditional branch logic of the virtual program to the underlying hardware conditional branch predictors. Second, it should reduce the redundant memory traffic back and forth to the interpreter's expression stack by holding temporary results in registers when possible. Third, it should support a few simple speculative optimizations.

Our JIT does not build any internal representation of a trace other that what is described in Section 6.4. Instead, it performs a single pass through each trace allocating registers and generating code. Register allocation is very simple. As we examine each virtual instruction we maintain a *shadow stack* which associates registers, temporary values and expression stack slots. Whenever a virtual instruction needs to pop an input we first check if there already is a register for that value in the corresponding shadow stack slot. If there is we use the register instead of generating any code to pop the stack. Similarly, when 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 stack but forgo generating any code to push the value. Thus, 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 [55].

Since traces contain no control merge points there is no additional complexity at trace exits other than the generation of the trace exit handler. As described in Section 6.6.1 trace exit handlers include generated code that flushes all the values in registers to the expression stack in preparation for execution returning to the interpreter. This is done by walking the shadow stack and storing each slot that is not already spilled into its home location. Consequently, the values stay in registers if execution remains on-trace, but are flushed when a trace exit occurs.

Linked trace exits result in potentially redundant stack traffic as values are flushed by the trace exit handler only to be reloaded by the destination trace.

Similar to a trace exit handler, 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, a sequence of calls is generated to dispatch the bodies of the uncompilable virtual instructions. Compilation resumes, with an empty shadow stack, with any compilable virtual instructions that follow. This means that generated code must be able to load and store values to the same Java expression stack referred to by the C code implementing the virtual instruction bodies. Our current PowerPC implementation side-steps this difficulty by dedicating hardware registers for values that are shared between generated code and bodies. Currently 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 above for trace exit handlers.

The actual machine code generation is performed using the ccg [54] run-time assembler.

6.6.3 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 One property of the trace selection heuristic is that innermost loops of a program are often selected into a single trace with the reverse branch at the end. (This is so 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.) Thus, when the trace is generated the loop will be obvious because the trace will end with a virtual branch back to its beginning. This seems an obvious optimization opportunity that, so far, we have not exploited other than to compile the last trace exit as a conditional branch back to the head of the trace.

Virtual Method Invocation When a trace executes, if the class of the invoked-upon object is different than when the trace was generated, a trace exit must occur. At trace generation time we know the on-trace destination of each call and from the training profile 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 Java run time 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. The virtual guard is an example of a trace exit that guards a speculative optimization [30].

Inlining The final optimization we will describe is a simple form of 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 generator 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) and it cannot throw. Then, no invoke is necessary 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 run time stack exactly as the (optimized away) invoke would have done

6.7 Polymorphic bytecodes

So far we have implemented our ideas in a Java virtual machine. However, we expect that many of the techniques will be useful in other virtual machines as well. For instance, languages like Tcl or JavaScript define polymorphic virtual arithmetic instructions. An example would be ADD, which adds the two values on the top of the expression stack. Each time it is dispatched

ADD must check the type of its inputs, which could be integer, float or even string values, and perform the correct type of arithmetic. This is similar to polymorphic method invocation.

We believe the same profiling infrastructure that we use to optimize monomorphic callsites in Java can be used to improve polymorphic arithmetic bytecodes. 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. Now, suppose that an ADD in Tcl is effectively monomorphic. Then, we would generate two virtual guards, one for each input. Each would check that the type of the input is the same as observed during training and trace exit if it differs. Then, we would dispatch a type-specialized version of the instruction (integer ADD, float ADD, string ADD, etc) and/or generate specialized code for common cases.

6.8 Other implementation details

Our use of a dispatch loop similar to Figure 6.1 in conjunction with ending virtual bodies with inlined assembler return instructions results in a control flow graph that is not apparent to the compiler. This is because the optimizer cannot know that control flows from the inlined return instruction back to the dispatch loop. Similarly, the optimizer cannot know that control can flow from the function pointer call in the dispatch loop to any body. We insert computed goto statements that are never actually executed to simulate the missing edges. If the bodies were packaged as nested functions like in Figure 4.1 these problems would not occur.

6.9 Packaging and portability

A obvious packaging strategy for a portable language implementation based on our work would be to differentiate platforms into “primary” targets, (i.e those supported by our trace-oriented JIT) and “secondary” targets supported only by direct threading.

Another approach would be to package the bodies as for subroutine threading (i.e. as illustrated by Figure 4.2) and use direct call threading on all platforms. In Section 7.2 we show that although direct call threading is much slower than direct threading it is about the same speed as switch dispatch. Many useful systems run switch dispatch, so presumably its performance is acceptable under at least some circumstances. This would cause the performance gap between primary and secondary platforms to be larger than if secondary platforms used direct threaded dispatch.

Bodies could be very cleanly packaged as nested functions. Ostensibly this should be almost as portable as the computed goto extensions direct threading depends upon. However nested functions do not yet appear to be in mainstream usage and so even gcc support may be unreliable. For instance, a recent version of gcc, version 4.0.1 for Apple OSX 10.4, shipped with nested function support disabled.

Chapter 7

Evaluation of Yeti

In this chapter we show how Yeti gradually improves in performance as we extend the size of execution units. We prototyped Yeti in a Java VM (rather than a language which does not have a JIT) to allow comparisons of well-known benchmarks against other high-quality implementations.

In order to evaluate the effectiveness of our system we need to examine performance from three perspectives. First, we show that almost all execution comes from the trace cache. Second, to evaluate the overhead of trace selection, we measure the performance of our system with the JIT *turned off*. We compare elapsed time against SableVM and a version of JamVM modified to use subroutine threading. Third, to evaluate the overall performance of our modest trace-oriented JIT compiler we compare elapsed time for each benchmark to Sun's optimizing HotSpot™ Java virtual machine.

Table 7.1 briefly describes each SpecJVM98 benchmark [62]. We also report data for `scimark`, a typical scientific program. Below we report performance relative to the performance of either unmodified JamVM 1.3.3 or Sun's Java Hotspot JIT, so the raw elapsed time for each benchmark appears in Table 7.1 also.

All our data was collected on a dual CPU 2 GHz PPC970 processor with 512 MB of memory running Apple OSX 10.4. Performance is reported as the average of three measurements

Table 7.1: SPECjvm98 benchmarks including elapsed time for unmodified JamVM 1.3.3 and Sun Java Hotspot 1.05.0_6_64

Benchmark	Description	Elapsed Time (seconds)	
		JamVM	Hotspot™
compress	Lempel-Ziv compression	98	8.0
db	Database functions	56	23
jack	Parser generator	22	5.4
javac	Java compiler JDK 1.0.2	33	9.9
jess	Expert shell System	29	4.4
mpeg	decompresses MPEG-3	87	4.6
mtrt	Two thread raytracer	30	2.1
raytrace	raytracer	29	2.3
scimark	FFT, SOR and LU, 'large'	145	16

of elapsed time, as printed by the `time` command.

Java Interpreters We present data obtained by running various modifications to JamVM version 1.3.3 built with gcc 4.0.1. SableVM is a JVM built for quick interpretation. It implements a variation of selective inlining called *inline threading* [29]. SableVM version 1.1.8 has not yet been ported to gcc 4 so we compiled it with gcc 3.3 instead.

7.1 Effect of region shape on region dispatch count

For a JIT to be effective, execution must spend most of its time in compiled code. 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 trace cache. In our system, regions of generated code are called from dispatch loops like those illustrated by Figures 3.2 and 6.1. In this section, we report how many iterations of the dispatch loops occur during the execution of each benchmark. Figure 7.1 shows how direct call threading (DCT) compares to basic blocks (BB), traces with no linking (TR) and linked traces (TR-LINK). Note the y-axis has a logarithmic scale.

DCT dispatches each virtual instruction independently, so the DCT bars on Figure 7.1

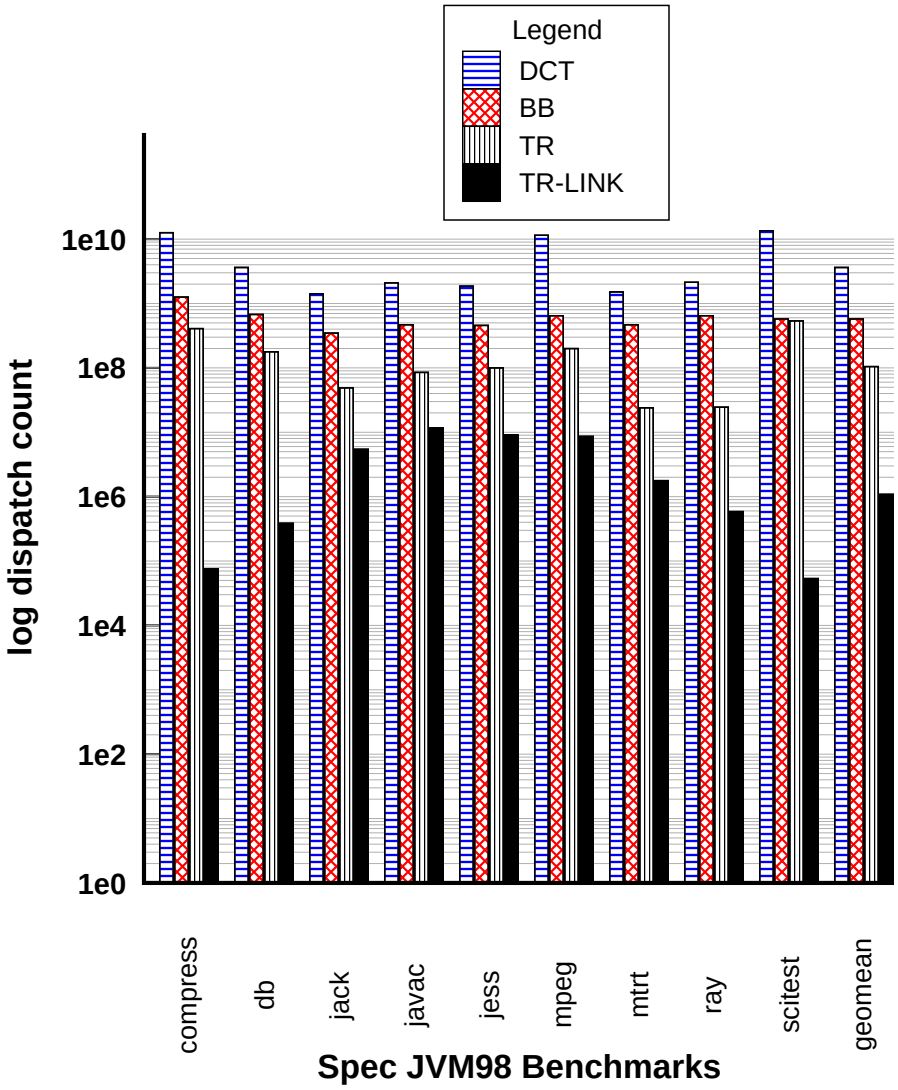


Figure 7.1: Log number of dispatches executed vs region shape.

report how many virtual instructions were executed. Comparing the geometric mean across all benchmarks, we see that BB reduces the number of dispatches relative to DCT by about a factor of 6.3. For each benchmark, the ratio of DCT to BB shows the dynamic average basic block length. As expected, the scientific benchmarks have longer basic blocks. For instance, the dynamic average basic block in `scitest` has about 20 virtual instructions whereas `javac`, `jess` and `jack` average about 4 instructions in length.

Even without trace linking, the average dispatch of a trace causes about 10 times more virtual instructions to be executed than the dispatch of a BB. (This can be read off Figure 7.1 by dividing the height of the TR geomean bar into the BB geomean bar.) This shows that traces do predict the path taken through the program. The improvement can be dramatic. For instance, while running TR, `javac` executes about 22 virtual instructions per trace dispatch, on average. This is much longer than its dynamic average basic block length of 4 virtual instructions.

TR-LINK makes the greatest contribution, reducing the number of times execution leaves the trace cache by between one and 3.7 *orders of magnitude*. The reason TR-LINK is so effective is that it links traces together around loop nests.

Although these data show 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 [9], remains future work. Nevertheless, we can relate a few anecdotes based on data that our profiling system collects. For instance, we observe that for an entire run of the `compress` benchmark all generated traces contain only 60% 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. On the other hand, for `javac` we find that the traces bloat – almost eight *times* as many virtual instructions appear in traces than are contained in the loaded methods. Improvements to our trace selection heuristic, perhaps adopting the suggestions of Hiniker et al [37], are future work.

7.2 Effect of region shape on performance

Figure 7.2 shows how performance varies as differently shaped regions of the virtual program are identified, loaded and dispatched. The figure shows elapsed time relative to the elapsed time of the unmodified JamVM distribution, which uses direct-threaded dispatch. Our compiler is turned off, so in a sense this section reports the dispatch and profiling overhead of Yeti by comparing to the performance of other high-performance interpretation techniques. The four bars in each cluster represent, from left to right, subroutine threading (SUB), direct call threading (DCT), basic blocks (BB), unlinked traces (TR), and linked traces (TR-LINK).

The simplest technique, direct call threading, or DCT, dispatches single virtual instruction bodies from a dispatch loop as in Figure 3.2. As expected, DCT is slower than direct threading by about 50%. Not shown in the figure is switch dispatch, for which the geometric mean elapsed time across all the benchmarks is within 1% of DCT. DCT and SUB are baselines, in the sense that the former burdens the execution of every virtual instruction with the overhead of the dispatch loop, whereas for the latter, all overhead was incurred at load time. The results show that SUB is a very efficient dispatch technique [8]. Our interest here is to assess the overhead of BB and TR-LINK by comparing them with SUB. BB discovers and generates code at runtime that is very similar to what SUB generates at load time, so the difference between them is the overhead of our profiling system. Comparing the geometric means across benchmarks we see that BB is about 43% slower than SUB. On the other hand, it is difficult to move forward from SUB dispatch, primarily because it is hard to add and remove the profiling needed for dynamic region selection.

Execution of TR-LINK is faster than BB primarily because trace linking so effectively reduces dispatch loop overhead, as described in Section 7.1. We have not yet investigated the micro-architectural reasons for the speedup of TR-LINK compared to SUB. Presumably it is caused by the same factors that make context threading faster than SUB [8], namely helping the hardware to better predict the destination of virtual branch instructions. Regardless of the precise cause, TR-LINK more than makes up for the profiling overhead required to identify and

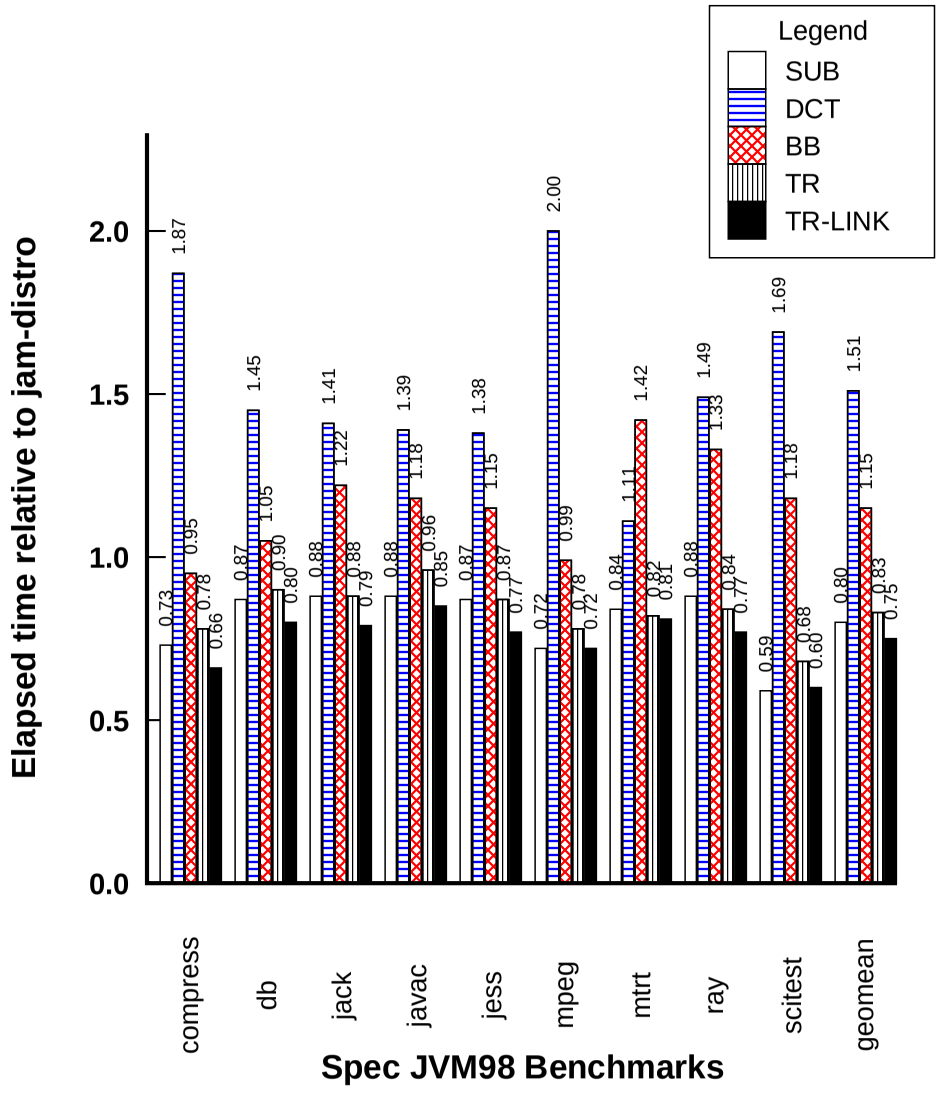


Figure 7.2: Elapsed time of Yeti *JIT disabled* relative to unmodified direct threaded JamVM version 1.3.3.

generate traces. In fact, even before we started work on our JIT, our profiling system already ran faster than SUB. Looking forward to Figure 7.3, we see that TR-LINK outperforms selective inlining as implemented by SableVM 1.1.8 as well.

For all benchmarks, performance improves as execution units become longer, that is, BB performs better than DCT, TR performs better than BB, etc. Our approach is indeed allowing us to gradually improve performance by gradually investing in better region selection.

7.2.1 JIT Compiled traces

Figure 7.3 compares the performance of our best-performing version of Yeti (JIT), to SableVM (SABVM). Performance is reported relative to the Java HotSpot™ JIT. In addition, we show the TR-LINK condition from Figure 7.2 again to relate our interpreter and JIT performance. In most cases TR-LINK, our profiling system alone (i.e without the JIT), does as well or better than SableVM. `Scitest` and `mpeg` are exceptions, where SableVM's implementation of selective inlining works well on very long basic blocks.

Not surprisingly, the optimizing HotSpot™JIT generates much faster code than our naive compiler. This is particularly evident for mathematical and heavily looping codes like compress, mpeg, the raytracers and scitest. Nevertheless, despite supporting only 50 integer and object virtual instructions, our trace JIT improves the performance of integer programs like `compress` significantly. Our most ambitious optimization, of virtual method invocation, improved the performance of `raytrace` by about 32%. `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 the accessor methods. Comparing geometric means, we see that our trace-oriented JIT is roughly 24% faster than just linked traces.

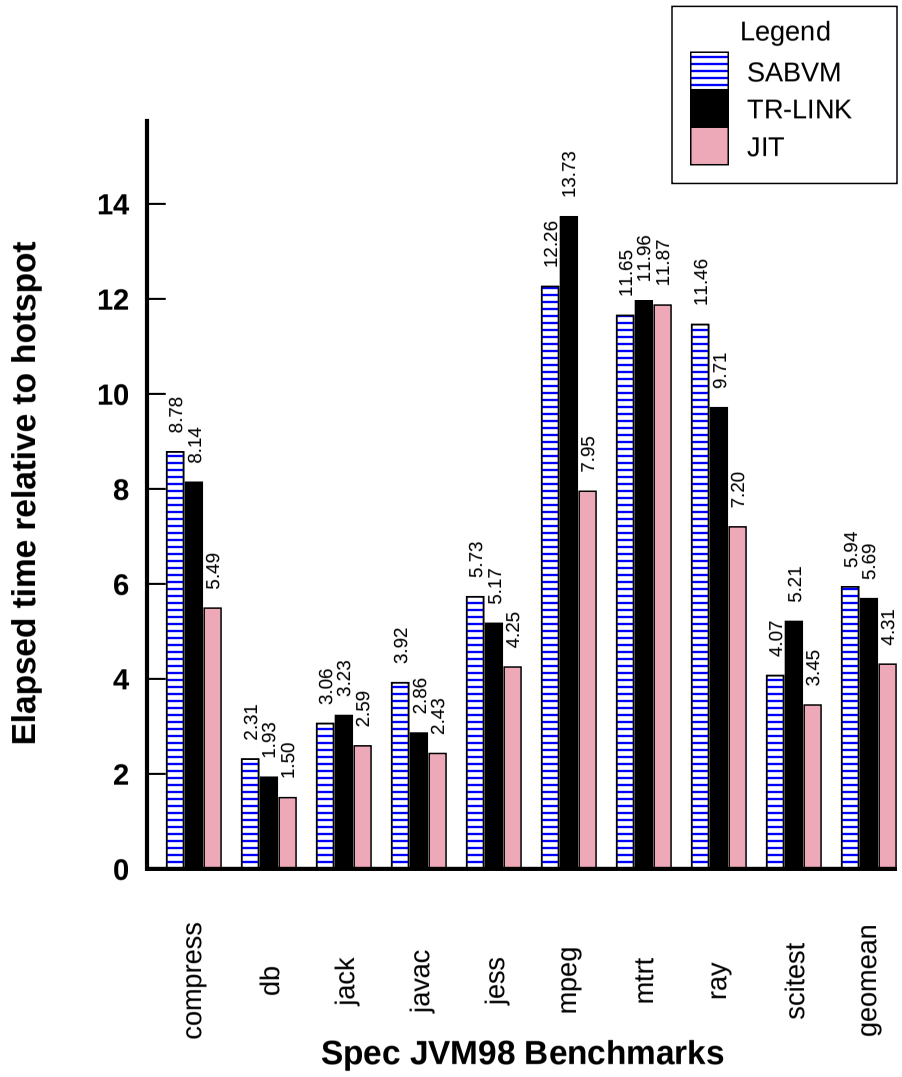


Figure 7.3: Elapsed time relative to Sun Java 1.05.0_6_64 compares JIT, our JIT-enabled version of Yeti, vs SableVM 1.1.8 with selective inlining.

Chapter 8

Conclusions and Future Work

We described an architecture for a virtual machine interpreter that facilitates its gradual extension to a trace-based mixed-mode JIT compiler. We start by taking a step back from high-performance dispatch techniques to direct call threading. We package all execution units (from single instruction bodies up to linked traces) as callable routines that are dispatched via a function pointer in an old-fashioned dispatch loop. The first benefit is that existing bodies can be reused by generated code, so that compiler support for virtual instructions can be added one by one. The second benefit is that it is easy to add instrumentation, allowing us to discover hot regions of the program and to install new execution units as they reveal themselves. The cost of this flexibility is increased dispatch overhead. We have shown that by generating larger execution units, the frequency of dispatch is reduced significantly. Dispatching basic blocks nearly breaks even, losing to direct threading by only 15%. Combining basic blocks into traces and linking traces together, however, wins by 17% and 25% respectively. Investing the additional effort to generate non-optimized code for roughly 50 integer and object bytecodes within traces gains an additional 18%, now running nearly twice as fast as direct threading. This demonstrates that it is indeed possible to achieve gradual, but significant, performance gains through gradual development of a JIT.

Substantial additional performance gains are possible by extending the JIT to handle more

types of instructions such as the floating point bytecodes, and by applying classical optimizations such as common subexpression elimination. More interesting, however, is the opportunity to apply dynamic and speculative optimizations based on the profiling data that we already collect. The technique we describe for optimizing virtual dispatch in Section 6.6.3 could be applied to guard various speculations. In particular, this technique could be used in languages like Python or JavaScript to optimize virtual instructions that must accept arguments of varying type. Finally, just as basic blocks are collected into traces, so traces can be collected into larger units for optimization.

The techniques we applied in Yeti are not specific to Java. By lowering the up-front development effort required, a system based on our architecture can gradually bring the benefits of mixed-mode JIT compilation to other interpreted languages.

Chapter 9

Remaining Work

We believe that our research is mostly complete and that we have shown that our efficient interpretation technique is effective and supports a gradual extension to mixed-mode interpretation. By modestly extending our system and collecting more data we can more fully report on the strenghts and weaknesses of our approach. Hence, during the winter of 2007 we propose to extend the functionality and performance instrumentation of our JIT compiler. These extensions and related data collection and writing-up should, if accepted by the committee, allow the dissertation to be finished by late spring or early summer of 2007.

The remaining sections of this chapter describe work we intend to pursue.

9.1 Compile Basic Blocks

In the push to compile traces we skipped the obvious step of compiling basic blocks alone. The basic block region data presented in Chapter 7 is for CT-style basic blocks with no branch inlining. It would be interesting to compare the performance of basic blocks compiled with our JIT to traces. Especially on loop nest dominated programs with long basic blocks, like scimark, compiled basic blocks might perform well enough to recoup the time spent compiling cold blocks.

9.2 Instrument Compile Time

Our infrastructure does not currently make any attempt to record time spent compiling. Since compiling short traces will take much less time than the resolution of the Unix clock some machine dependent tinkering may be required. Knowing the overhead of compilation would help characterize the overhead of our technique.

9.3 Another Register Class

Adding support for float registers would make our performance results for float programs like scimark more directly comparible to high performance JIT compilers like HotSpot. Extending our simple JIT to handle another register class would show that our design is not somehow limited to one register class. Compiler support would need to be extended by about another dozen floating point virtual instructions in order to test our design.

9.4 Measure Dynamic Proportion of JIT Compiled Instructions

As the JIT is extended to support for more virtual instructions it would be useful to measure the proportion of all executed virtual instructions made up by JIT compiled instructions.

Bibliography

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

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

[3] 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.

[4] 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`.

[5] 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`.

[6] 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.

[7] 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`.

[8] 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.

[9] 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`.

[10] 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.

[11] 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`.

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

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

[14] 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`.

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

[16] 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`.

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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`.

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

[26] 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.

[27] 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.

[28] 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`.

[29] 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.

[30] 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.

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

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

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

[34] 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.

[35] 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`.

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

[37] 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.

[38] 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`.

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

[40] 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`.

[41] 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`.

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

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

[44] 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`.

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

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

[47] 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.

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

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

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

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

[52] 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`.

[53] 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`.

[54] 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`.

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

[56] 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.

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

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

[59] 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.

[60] 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.

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

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

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

[64] 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`.

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

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

[73] 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`.

[74] 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.

[75] 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%`.

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

[77] 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.