Automatic Generation of Optimised Virtual Machine Interpreters

by

Mr Kevin Casey, BSc. MSc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Doctor of Philosophy

University of Dublin, Trinity College

September 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Mr Kevin Casey

February 1, 2006

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Mr Kevin Casey

February 1, 2006

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr David Gregg, whose enthusiasm and encouragement ensured the completion of this thesis.

I would also like to thank the members of the Computer Architecture Group who have been great company and of great assistance through the years. Particular thanks to Andrew Beatty, Andy Nisbet, Brian Davis and Yunhe Shi from the group, with whom I have worked on many issues relating to the Java Virtual Machine.

Special thanks are also due to M. A. Ertl of the Technical University, Vienna, for his unselfish assistance and ready availability to offer advice throughout the duration of this project, and also for acting as host when I visited Vienna in 2004.

Finally, my greatest appreciation is reserved for Rosemary, my family and my friends for the patience they have all shown as I disappeared into my work. This work would not have been completed without their boundless understanding.

Mr Kevin Casey

University of Dublin, Trinity College
September 2005

Automatic Generation of Optimised Virtual Machine Interpreters

Publication No. _____

Mr Kevin Casey, Ph.D.
University of Dublin, Trinity College, 2005

Supervisor: Dr. David Gregg

Virtual Machines (VMs) are commonly used as execution platforms for many modern high-level languages. Two important examples are the Java VM, intended for running Java applications, and Microsoft's Common Language Runtime (CLR), intended for executing .NET applications compiled from C#/VB.NET. Typical VMs are implemented as either interpreters or Just-In-Time compilers. Interpreters are slower but have several advantages such as reliability, portability and memory efficiency that make them ideal for certain types of application.

This thesis concentrates on various optimisations to improve the performance of VM interpreters. We show how it is possible to select and apply broadly useful optimisations, and even more importantly, show how these optimisations can be implemented

in a automatic and portable manner. In order to facilitate this work we develop an interpreter generation tool, Tiger which provides extensive support for these optimisations. Details of this tool are presented, along with a discussion of how it supports a number of optimisations.

A new optimised, portable JVM called Fastcore is constructed using Tiger, and then evaluated. The applied optimisations such as faster dispatch methods, constant inlining, conditional loading of operands and faster method dispatch are detailed along with their cumulative effect (an average speedup of 1.31) over an equivalent unoptimised JVM interpreter.

Remaining optimisations are then classified into two broad categories; static instruction enhancement and dynamic instruction enhancement. Static instruction enhancements are comprised of instruction replication, instruction concatenation (superinstructions) and instruction specialisation. We show how these optimisations can improve the performance of our optimised interpreter by a speedup of up to 2.1 when the interpreter is being optimised for a broad range of programs, and up to 3.35 when the interpreter is being customised for a particular program.

Dynamic instruction enhancements are comprised of dynamic instruction replication, along with a number of methods for creating dynamic superinstructions. The effects of these dynamic optimisations are examined in comparison to each other and against the static optimisations previously presented. These more generic dynamic optimisations improve performance by a speedup of up to 2.76 for a broad range of programs.

Extensive results using hardware performance counters are presented for all of these optimisations. Some surprising results are encountered, which are highlighted and explained. These results give greater insight into the behaviour of VM interpreters and help the construction of simpler, faster, more maintainable VM interpreters.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Virtual Machines and Interpreters

Virtual Machines (VMs) are commonly used as execution platforms for many modern high-level languages. Two important examples are the Java VM (JVM) [GJSB00], intended for running Java applications, and Microsoft's Common Language Runtime (CLR) [Sin03], intended for executing .NET applications compiled from C#/VB.NET. Most high-level conventional languages are compiled right down to the native code of the CPU on which they are executed. Similarly, languages intended for execution on virtual machines are compiled down to the native language of the machine they are to be executed on; in this case *virtual machine code*.

Running application code on a virtual machine rather than a native one has two main advantages [SN05]. Firstly, applications can be distributed in an architecture-neutral format. To run an application on another architecture, all that is required is to port the virtual machine. The work done in porting this one program to a new architecture allows any application code for the virtual machine to be run on that new architecture. Secondly, security checks can be implemented on a virtual machine that are too complex or expensive to implement in hardware. For example, the JVM has software checks to ensure that all variables are initialised before they are used. A similar check in hardware would be too complex to implement efficiently.

Virtual machines have other smaller advantages too. Because the instruction sets of virtual machines are idealised, and not the result of hardware constraints on a CPU,

they are often simpler. Such instruction sets are often high level, well organised code that is an easy target for compilation. For example, it can be simpler to write a source language to VM compiler and a simple implementation of the VM, rather than a source to native code compiler. VM instruction sets tend to be much coarser-grained than machine instructions, performing more work per instruction. This, coupled with the stack-based architecture that is currently prevalent in virtual machine design, means that the applications in virtual machine native code are quite compact. This compact code is ideal for running on memory constrained systems or for transferring quickly across a network.

In order to do its job, a virtual machine has to run virtual machine code. However, the way a virtual machine is implemented has significant effects other than on the ability to execute virtual machine code. The principal aspects of virtual machine behaviour affected by implementation details are, unsurprisingly, speed and memory requirements.

Like real machines, virtual machines have architectures which include components responsible for memory management, program loading and execution. Of all these components, the component responsible for the execution of the virtual machine code has the greatest effect on both speed and memory requirements. This component, the *execution engine* is commonly implemented in one of two ways: dynamic compilation or interpretation.

Dynamic compiling execution engines are generally the fastest way to implement a VM (apart from execution in hardware) [Arm98]. These engines compile VM code to native machine code before execution. The most common strategy is to compile each method in the program the first time it is executed. Such Just-In-Time (JIT) compilers are often large, complex applications, particularly if they perform many optimisations. Typically, JIT compilers are not quite as efficient as native compiled code, because:

1. there is some overhead in compiling the code and
2. JIT compilers have to be fast, so they perform fewer optimisations than regular compilers.

Nonetheless, JIT compilers approach the speed of compiled C code [LN04], and are typically around ten times faster than an efficient interpreter.

Interpreter-based execution engines act as an execution engine in a real CPU might act, namely by a continuous fetch-execute cycle until the virtual machine terminates. These execution-engines, or VM interpreters, are slower because they must incur the cost of the instruction fetch and the cost of jumping to the code responsible for executing that virtual machine instruction. However, despite this speed penalty, interpreter-based virtual machines have several advantages [MB99]:

- Interpreters do not require much memory. They can make significant memory savings, because they operate directly on the compact virtual machine code without translation to native code. This makes them an ideal choice where memory is scarce, for example embedded systems.

- Interpreters are easy to port. Most interpreter instructions are quite simple, and porting those instructions to a new architecture is usually straightforward. A dynamically compiling virtual machine is inherently intertwined with the architecture it is compiling native code for.

- Interpreters are smaller and less complex programs than JIT compilers. As a result it is easier to be confident that they are correct and they are easier to maintain.

- Interpreters make it easier to provide VM-level programmer tools such as tracers, profilers and debuggers. These tools are readily available for many popular interpreters. This is due to the direct relationship between VM code and the structure of the interpreter.

- Interpreters can actually be faster than dynamic compilers for some sections of code. In particular, large sequences of infrequently executed code can be executed more efficiently, because the cost of compilation before execution is not incurred. Some dynamic compilers, such as Sun's HotSpot JVM [Gri98], use interpreters for certain regions of code to avoid compilation overhead.

In this research, we examine various possibilities for improving VM-interpreter performance. To implement many of these optimisations we build an interpreter generation tool, Tiger. This tool not only allows the implementation of these optimisations, but also permits parameters of those optimisations to be changed easily.

1.2 Our Thesis

Two of the greatest benefits of interpreters are portability and simplicity. Perhaps their greatest limitation is performance. Our thesis is that interpreters can be substantially optimised in a platform general manner, and furthermore, that these optimisations can be:

- Portable - by applying optimisations at source-code level.
- Simple for the interpreter writer - by a domain-specific language for specifying interpreter cores and a tool for generating the cores from that specification.

Thus, we show how performance can be improved considerably without sacrificing the benefits of portability and simplicity.

1.3 Contributions

- *Tiger, a tool for building interpreters, applying optimisations and investigating their effects.*

Our first contribution to research into interpreter optimisation is the Tiger interpreter generator, implemented in Java. This generator uses a domain-specific language to solve a domain specific problem; namely interpreter construction and optimisation. A wide range of optimisations are supported by this tool, which also permits the debugging and profiling of running Java programs.

- *Proving the effectiveness of portable source-code optimisations in an interpreter.*

We prove that it is possible to optimise a Java interpreter using only high-level source code optimisations which are highly portable. To do this, a less-optimised Java VM is selected and its interpreter core is replaced with a new one that is constructed using Tiger. Only high level portable source code optimisations are applied to this new interpreter, rather than architecture-specific machine code optimisations. Results for the new optimised interpreter, Fastcore, are presented and compared against a hand-tuned highly-optimised JVM interpreter.

- *Examining the effect of a number of static instruction enhancements to the Java interpreter.*

A number of optimisations which change the standard internal instruction set of the Java interpreter are detailed. The effects these optimisations have on standard benchmarks are examined using hardware performance counters. A direct threaded interpreter is used to allow our interpreter to work with a larger set of instructions than the 256 instruction limit inherent in bytecode, adding up to 512 extra instructions to the JVM core at any one time.

- *Implementing and examining the effect of a wide range of dynamic code copying optimisations for Java.*

Using dynamic code copying techniques to improve interpreter performance has received much interest since Piumarta et al [PR98] presented dynamic inlining of executable code. A number of variations of the basic technique are implemented and evaluated and a detailed comparison and discussion follow. As dynamic code copying is a technique that is similar in some ways to a JIT compiler, a comparison of the two techniques is also supplied, both in terms of execution speed and memory requirements.

1.4 Collaborations

During the course of this research the author of this thesis has collaborated with several colleagues. Andrew Beatty provided early support in terms of obtaining and setting up suitable benchmarks for evaluating JVMs. He also obtained and compiled an off-the-shelf JVM that nonetheless required some changes in order to work correctly. His contributions are recognised in his joint authorship of [BCGN03]. Anton Ertl provided much expertise and advice in addition to the vmgen tool that was the primary inspiration for the Tiger tool and some of the optimisations it provides for. His contributions are recognised in his joint authorship of [CGE05a, CGE05b, CGEN03]. He also collaborated with the author in the writing of a paper submitted to ACM Transactions on Programming Languages and Systems (TOPLAS). This paper is composed of two sets of work relating to optimisation. One section, relating to the JVM was implemented, tested and documented by the author. The other section, relating to GForth is based

on work done on virtual machines for the Forth language by Anton Ertl and David Gregg.

Both Andrew Nisbet and David Gregg provided invaluable advice, assistance and guidance throughout the project. Their contributions are recognised in joint authorship of [BCGN03, CGEN03] and [BCGN03, CGE05a, CGE05b, CGEN03] respectively.

Finally, the author provided assistance in the initial stages of the implementation of a register-machine JVM. His work is recognised in joint authorship of [DBC+03].

1.5 Overview

The remainder of this thesis is structured as follows:

Chapter 2 This chapter examines the history of Java. The increased popularity of embedded devices as Java platforms is identified as a trend that has renewed interest in efficient interpreters. The Java Virtual machine and its components are introduced and explained. A number of execution engine types are also introduced, the most pertinent being the interpreter. In this chapter, pipelining and branch prediction are singled out as important factors in an interpreter's performance.

Chapter 3 This chapter highlights some past and present work by others pertaining to the performance and optimisation of virtual machine interpreters. Particular attention is paid to work relating to the cost of dispatches in an interpreter and optimisations designed to reduce those costs. Work relating to a number of optimisations which reduce memory accesses during bytecode execution is also examined.

Chapter 4 This chapter presents the **T**rinity **I**nterpreter **Ge**nerato**r** (Tiger). This tool, developed by the author, allows the automatic creation of an interpreter core from a user-supplied specification file. This specification file can include a selection of optimisations which will be applied to the Tiger-generated interpreter core. The chapter introduces the tool, presents some important implementation details and discusses some of the functionality of Tiger.

Chapter 5 In this chapter, the creation of a new optimised interpreter, Fastcore, is documented. This new interpreter is created by extensively modifying a less optimised interpreter, CVM. A new interpreter core is created in the Tiger tool, and selected optimisations are applied. This chapter discusses the process, the optimisations and examines the performance of the new optimised interpreter in comparison to some of its peers.

Chapter 6 This chapter presents a number of interpreter instruction enhancements designed to improve runtime performance. Specifically, three classes of instruction enhancement are examined; specialisation, replication and superinstructions. For each class of enhancement, the methods used to select new instructions to be added to the interpreter are discussed. Important implementation details for each enhancement, many relating to so-called *quickable instructions*, are detailed. Performance measurements are presented for each enhancement, and certain counter-intuitive results are obtained and explained.

Chapter 7 This chapter presents a selection of dynamic instruction enhancements. The chapter begins with an explanation of the advantages of dynamic enhancements over static enhancements. For each class of dynamic enhancement, the basic method is explained and important implementation issues for that enhancement class are discussed. Results are presented for the various types of enhancement, and close attention is paid to the instruction cache performance and additional memory requirements of these enhancements. The chapter concludes with an objective comparison between these optimisations and those provided for by Just-In-Time compilers.

Chapter 8 In the last chapter, the results of the thesis are summarised, highlighting some of the most notable contributions. Finally, we identify some interesting aspects arising from the work that warrant further research.

Chapter 2

Background

2.1 Introduction

Eleven years after its inception, the use of Java [GJSB00] has become commonplace. The number of Java-enabled devices is currently estimated at 2.5 billion and the number of worldwide Java developers at 4.5 million. The nature of devices on which Java is deployed has shifted recently towards embedded devices. At present the number of Java-enabled handset devices exceeds the number of Java-enabled desktop PCs [LS05].

2.2 A Brief History of Java

Arguably the most attractive feature of Java, its portable *virtual machine* nature, has been around in various guises for many years. This virtual machine is a program that isolates an application from the hardware it is running on. To execute, the application makes use of services provided by the virtual machine. This approach assists in portable code, at least for the application developer. In order to support the same set of services in a standardised way on several platforms, the virtual machine may have to be completely rewritten. This does not concern the application developer, whose application will run on any virtual machine that provides a standardised set of services.

Apart from the portability issue, another strength of virtual machines is that the hardware resources of a machine can be shared among several users, each literally getting their own virtual machine. This was the motivation for much of the early work

done in virtual machines by IBM in the 1960s in the development of IBM CP/CMS operating system [Cre81] for the IBM 360.

Interpretive compiling, a process used by Java, is the method by which source code for an application intended for running on a virtual machine is compiled into an intermediate representation. This intermediate representation can then be executed directly by the virtual machine. In a sense it is a compiler, not with the language of a hardware machine as the target, but the language of the virtual machine as the target.

The first serious portable, interpretive compiler system evolved from the system BCPL [Ric71] (a forerunner to the C language) in 1971. OCODE [HSS80] the assembly-like intermediate language outputted by the Cambridge BCPL compiler was intended to be run on the BCPL Virtual Machine, a stack-based VM. A few years later in 1976, the UCSD P-system [BGCS82] was developed. This incorporated the Pascal-P language which was compiled by the Pascal-P compiler into intermediate bytecode called P-Code. Virtual machines existed for a number of architectures including the 6502, the 8080, the Z-80, and the PDP-11. As a result of this unprecedented portability it was quickly adopted by users of these relatively new architectures. Of all the forerunners to Java, it is the Pascal-P system with its portable P-code and stack-based VM that most closely resembles the interpretive compiler system that Java uses. In a sense it should be no surprise that such systems preceded Java so long ago. In the past, hardware constraints made interpreters quite popular due to the compact intermediate representation of application code. Indeed, these same hardware constraints are shared by many current embedded devices such as mobile phones and smart cards.

This embedded market was the target of Sun Microsystem's Oak programming language [Sun05a]. This language was designed from the ground up as an object oriented language for embedded devices. As such, common security and reliability issues were important design constraints. This resulted in the elimination of multiple inheritance, operator overloading and pointers. Renamed to Java for trademark reasons, and under the control of the FirstPerson project at Sun, the language could not find acceptance in the consumer electronics industry. As a result, the FirstPerson project was dissolved by Sun in 1994.

Despite this, a few members of the project continued their efforts in finding a market for Java. This changed when Bill Joy made the decision to get Java running inside WWW browsers. Even though it was not the original target market for Java, the

features of Java made it a perfect fit for the World Wide Web. Seeing Joy's work, Sun Microsystems realised this, supported the project and announced Java and HotJava, a Java-based Web browser in 1995 [Als95]. Netscape Inc [Ano95] and Microsoft [Lie95], followed suit by announcing support for Java in their respective browsers, cementing Java's place on the Internet.

In the last few years, Java has essentially returned to its roots, being deployed on many of the consumer devices for which it had been originally designed. Embedded devices such as mobile phones, smart cards, car navigation systems, gaming systems now run Java Virtual Machines as part or all of their processing functions.

2.3 The Java Virtual Machine

The Java system, at its most basic level consists of the Java compiler for compiling Java source code into bytecode, a Java Virtual Machine (JVM) for running that bytecode on a particular architecture, and the Java Class Libraries, a standard set of well documented libraries. Figure 2.1 illustrates the relationship between Java source, Java bytecode, the JVM and underlying architecture.

In Figure 2.1, a simple program Hello.java is compiled by a Java compiler. The compiler outputs a Hello.class file containing the bytecodes, the machine code as it were, for the Java Virtual Machine. These class files can be executed by a JVM on the same machine upon which compilation took place, or they could just as easily be transmitted across a network. They can even be run on a machine with a different architecture as long as that machine has a JVM. No matter what JVM the Hello.class file is run on, that JVM should have access to the standard Java Class Library.

When the Hello.class program is being executed, any external code dependencies are loaded, typically from the Java Class Library. This external code is usually also in the form of class files[1]. This process, an extended form of dynamic linking, is called *dynamic class loading*. In combination with the bytecode format and the standardised class library, this dynamic linking approach is one of the reasons Java class files are so compact. This compactness is one of the attributes of Java bytecode that lends itself to fast transmission through networks. As we saw in Section 2.2, this is a desirable

[1]These class files may or may not be stored in compressed form.

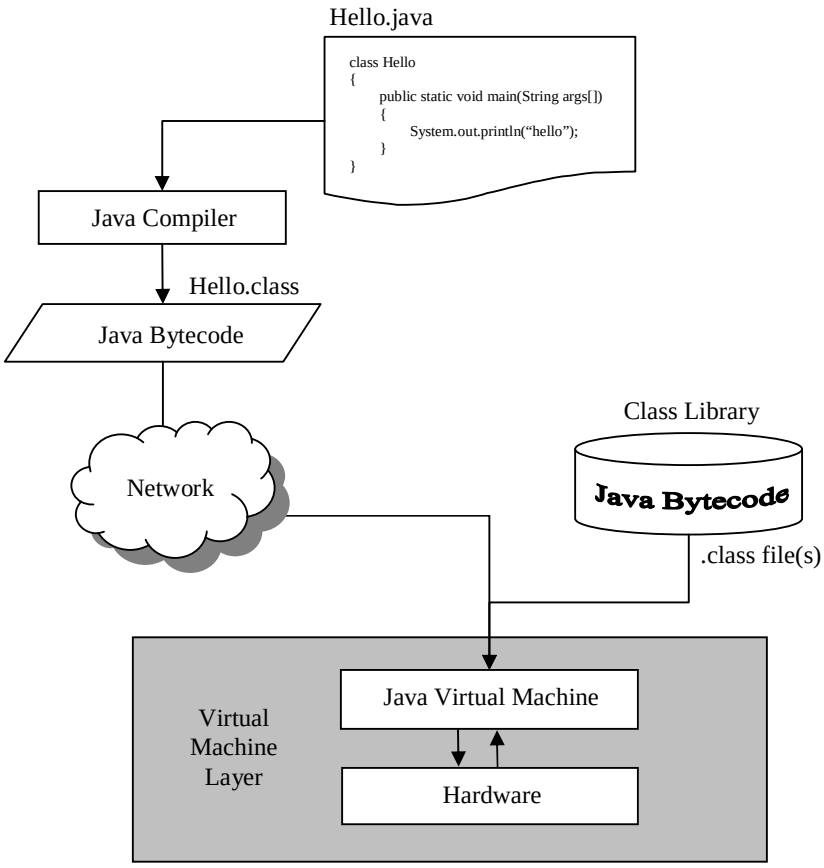


Figure 2.1: Overview of the Java System

property of Java in its role of supplying and running dynamic web content for browsers[2].

The bytecode itself is a stream of bytes, where each opcode is one byte long. Thus there is a maximum of 256 opcodes numbered from 0-255. Each instruction in the bytecode has a defined number of operands, also composed of bytes. Operands longer than a byte, for example integers, are either spread over several bytes or stored in the *constant pool*[3]. A limited number of types can appear as operands in the bytecode stream. These are the seven so-called primitive types in Java; byte, short, int, long, float double and char. Big endian ordering is used for operands that are split over several bytes. In the bytecode, there is no explicit type information encoded with operands. The number and type of operands in an instruction is implied from the instruction's opcode itself. For example the `iflt` (if less than zero) instruction is followed by two operand bytes representing a single signed 16 bit integer.

In Figure 2.1 the JVM and underlying hardware can often be viewed as a single machine designed for running Java, hence the grouping of the two logically into the Virtual Machine Layer. When providing support for Java on a new architecture, the only component that needs to be modified/rewritten is the only hardware dependant part of the whole package, namely this JVM.

The JVM, the subject of this thesis, is typically written in C/C++ perhaps with some assembly language for performance reasons. Regardless of the implementation language, or the underlying architecture, JVMs have a universal structure, based on the various roles it must carry out in its lifetime. Figure 2.2 illustrates this high-level structure of a typical JVM.

2.3.1 Execution Engine

The main processing part of the JVM is the *Execution Engine*, responsible for the execution of Java bytecode. There are a number of (potentially overlapping) methods for implementing the Execution Engine:

Interpreter-based For embedded systems without a native Java instruction set, an interpreter-based execution engine is a good choice as it requires little memory

[2]The Hello.java program above compiles to 401 bytes using j2sdk 1.4.2. In contrast a 'C' program for doing the exactly same task compiles to 36,864 bytes on a Pentium 4 using MSVC 7.1.

[3]A repository for larger datatypes, stored outside the bytecode and indexed from the bytecode wherever required. This approach yields more compact bytecode at the expense of run-time efficiency.

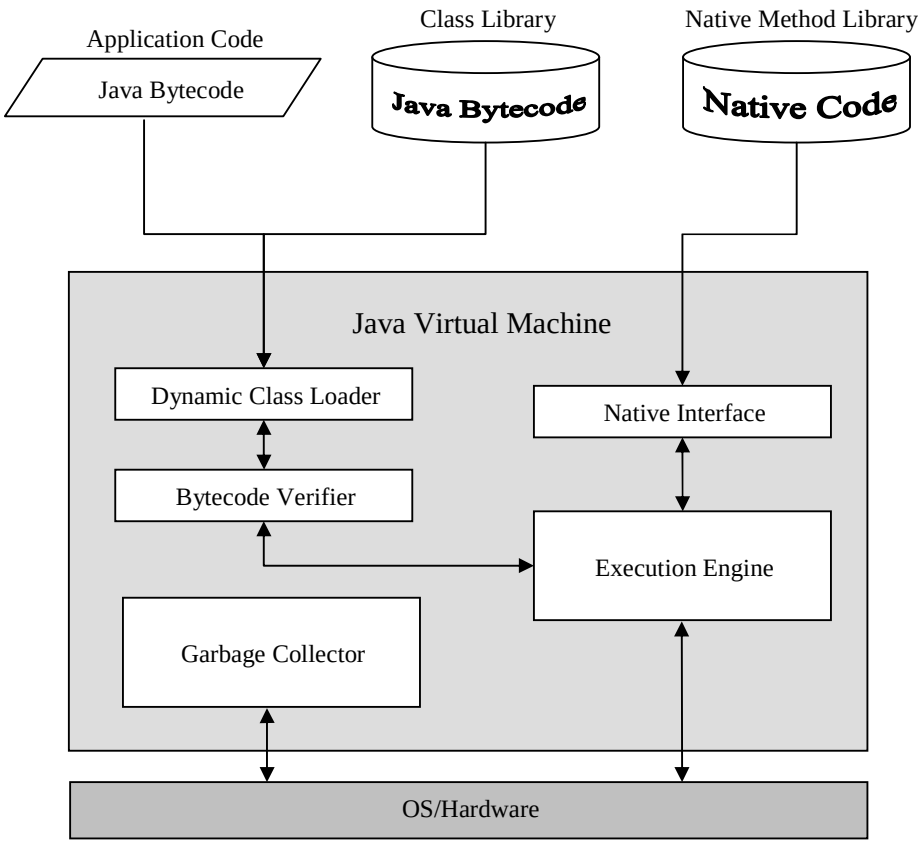


Figure 2.2: The Java Virtual Machine Structure

during the execution of Java bytecode. The interpreter reads the bytecode instruction for an instruction code (opcode). It then looks up a table to find the implementation code for this instruction code and then jumps (dispatches) to this code. When the code for the instruction is complete, the bytecode interpreter repeats the process beginning by reading the next instruction from the bytecode. While interpreter-based approaches are slow, they do have some notable benefits such as low run-time memory usage and are easy to port to new architectures. Because of this compact nature, Embedded JVMs such as the Java Card Virtual Machine [SZY00] are typically implemented as interpreters.

Ahead-Of-Time (AOT) compilation Ahead Of Time compilers, compile all the Java code to native code before the JVM begins execution. This is an attractive proposition where the bytecode to be run is known in advance. However because some Java applications use custom classloaders or load components at runtime, code from a fully-fledged AOT compiler must be augmented with a Java execution engine such as an interpreter or a Just-In-Time compiler. The Gnu Compiler for Java (GCJ) [GNU03] currently offers limited support for AOT compilation for Java source code. The applications compiled to native code by an AOT compiler will typically be much larger than the bytecode generated by a standard java compiler.

Just-In-Time (JIT) compilation By far the most popular technique for implementing an execution engine, this approach incorporates a Java to native code compiler into the running JVM. When a new bytecode method is encountered, the method may be converted to native code, just in time, i.e. just before it is executed. Because compilation to native code is expensive, some JITs only compile frequently executed regions of bytecode (*hotspots*). This has two implications. The first is that heuristics are needed to decide which regions of bytecode are going to be executed frequently and are worth compiling to native code. The second is that some bytecode will not get translated. This in turn implies that the JIT compiler must have an execution engine for dealing with these bytecodes. This secondary execution engine is usually implemented as an interpreter. Sun Microsystem's HotSpot JVM [Gri98] makes use of this dual-execution engine approach to combine execution of JIT compilation and bytecode interpretation in

an efficient manner.

CPU-based Java processors are processors with instruction sets and architectures that match or closely match the virtual machines upon which Java bytecode is to be run. Many, if not all, of the Java bytecode instructions are actually implemented in hardware. Because of this, on these systems java bytecode *is* native code. Some processors with Java support such as ARM processors with the Jazelle extension [Por04] have near-JVM architectures with most Java instructions being implemented in hardware and others being emulated by short sequences of processor native instructions. While Java processors are fast for executing Java bytecode, they are not as flexible as many other processors, and are more suitable for embedded devices rather than desktop/server machines.

2.3.2 Dynamic Class Loader

The job of the Dynamic Class Loader [GJSB00] is to load external classes, as and when they are required by the currently executing Java bytecode. These external classes can be in ROM, on a drive, or even in a remote location, accessed over a network. Once the appropriate class has been located, the dynamic class loader will verify that the class adheres to the class file format. The class file will then be loaded into the appropriate data structure in memory and the class in memory will be prepared and initialised. There can be multiple class loaders (implemented as classes) in Java. These classes themselves need to be loaded by a class loader. This implies the need for a root class loader, through which other class loaders can be loaded as the JVM executes. This root class loader is called the *primordial class loader*. It is typically written in native code and is platform dependent.

2.3.3 Bytecode Verifier

The Bytecode Verifier analyses the bytecode and checks for illegal activity. This enhances the security and the reliability of the JVM. In order to do its job, the bytecode verifier carries out a number of phased checks of loaded classes. The first phase happens immediately after the dynamic class loader has loaded a new class. This phase checks a number of items such as ensuring each opcode is valid and has the correct operands,

that the operand stack is always the correct height at each point in the bytecode, and that local variables are initialised before usage.

The next phase of the verification stage is often deferred until the individual bytecodes in the class are executed for the first time. This stage verifies symbolic references, resource identifiers for external classes, methods or fields. Many of these symbolic references may be references to classes that have yet to be loaded, so they must be loaded at this stage. When a symbolic reference is located, the bytecode verifier also checks that it contains the specified class, method or field, depending on the symbolic reference. If the target for the symbolic reference can be found and loaded into the JVM successfully, the corresponding symbolic reference can be replaced with a direct reference to the target. This will prevent further checking of the symbolic reference, since it has already been verified.

These relatively costly checks are carried out to satisfy Java's original intent to provide secure, robust computation for embedded devices and for virtual machines running in WWW browsers running code that may or may not be trusted.

2.3.4 Java Native Interface

The Java Native Interface [Lia99] is a part of the JVM which enables the JVM to call native code. This code can support platform dependant features not supportable through a JVM, pre-existing libraries that are too costly or unable to be ported to Java, or frequently executed tasks that would be better implemented as native code for performance reasons.

The JNI also has an important role, interfacing the JVM to JIT compilers. JIT compilers are built and supplied as standalone native code, platform dependant modules. The JVM uses the native interface to call the various functions of the just-in-time compiler. In addition, when methods have been compiled to native code by the JIT compiler, calls to these native methods can be performed through the native interface.

Using the JNI enables native code to create new Java objects and manipulate them. Native code can also access Java objects, even modifying Java objects that are passed as parameters. Similarly the JVM is able to inspect and modify objects created by native code using the JNI. The JNI also gives the native code the ability to catch, handle and throw exceptions, to load Java classes, to get information about classes

and to do runtime type checking.

2.3.5 The Garbage Collector

Each time the `new` operator in Java is used to create a new object, memory for that new object is allocated on the heap. In Java, unlike, for example, C++, memory is not explicitly de-allocated. This means that allocated memory that is no longer required must be freed through some other mechanism. This mechanism is the Garbage Collector [Ven00].

The Garbage Collector typically runs in its own thread and scans through the heap looking for objects that are no longer reachable. When such objects are found, the Garbage Collector removes these objects from memory. The advantages for this automatic allocation and deallocation of memory are a more secure JVM, and releasing the programmer from the labour and potential bugs associated with memory allocation errors. The disadvantage is that the garbage collection process consumes precious CPU time since it is executed at certain times during a JVM's execution.

The times when a Garbage Collector are to be run are not defined in the standard Java specification [GJSB00]. Indeed the Java specification does not even say how the Garbage Collector should do its job, merely that the heap must be garbage collected. This has led to a myriad of garbage collection algorithms being incorporated into JVMs, experimental or otherwise. Garbage collection algorithms vary from reference counting [Bev87] to more sophisticated Mark-Sweep algorithms [GSaC05] and copying algorithms such as the Semi-Space [Che70] and Generational [DKP00] algorithms. Differences between the various algorithms are in how they deal with memory fragmentation and how quickly they can find objects that require garbage collection.

Because no guarantees are made in the Java specification about the times when the Garbage Collector is invoked, nor how long each invocation should take, Java is often seen as unsuitable for real-time applications. The Real-Time Specification for Java (RTSJ) [BBG+00] provides for additional control over the Garbage Collector by the programmer. For example, it is possible to create threads called `NoHeapRealtimeThread`s that never require garbage collection. The programmer can also, where possible, determine for how long the current thread can be interrupted by the underlying Garbage Collector. Another possibility the programmer has is to force the Garbage Collector

to execute at certain times[4].

2.3.6 Threading Model

Threads are part of the Java standard, indeed, even in a single threaded Java application, the Garbage Collector must run as a separate thread. Therefore a compliant JVM must support threads. As with garbage collection, the mechanism of thread support is not specified by the Java standard. There are typically two choices, with some JVMs providing user selectable implementations of both:

1. Green threads (JVM-level threads) [New00]. In this approach, the entire JVM occupies a single OS thread. Threads of execution must share this single OS thread. This is sometimes called a many-to-one thread model.

2. Native threads. In this approach, the JVM can create multiple threads, one for each Java thread, and threads for the Garbage Collector as required. This is sometimes called a one-to-one threading model.

Both approaches have their strengths and drawbacks. Green threads cannot make use of Symmetric Multiprocessor (SMP) systems and can be inefficient as each time a blocking system call occurs, it must be wrapped in such a way that prevents the entire JVM from blocking. On the other hand, a JVM supporting native threads is less portable, but can exploit SMP. Additionally, the efficiency of native threads can be very high, depending on the OS threading implementation. When it comes to the creation, destruction or suspending of a Java thread, green threads still have the performance edge, since all these operations can be performed without a system call, unlike native threads.

For embedded systems, a green threads approach is normally the practice for two main reasons. Firstly, a JVM that uses green threads is more portable and secondly, the underlying OS (if indeed there is any) on an embedded system is unlikely to provide threading support. Because of this portability and the JVM-level fine grained control

[4]The original non-RTSJ Java standard allows for two calls, System.gc() or Runtime.gc(). However these are just recommendations from the programmer indicating that a garbage collection should happen here. Being merely recommendations, a standardised JVM can simply ignore them. This is not usually acceptable behaviour in real-time applications.

of Java threads, the green thread approach can make simpler the task of constructing and optimising a JVM.

2.4 Modern Processor Architecture

CPUs have evolved considerably from the days of simple fetch-decode-execute cycle based hardware. Various architectural improvements have been made to improve instruction throughput, even as the performance gap between memory and CPU widens. Some of these architectural aspects of the CPU have a surprisingly large effect on an interpreter's performance. The two main aspects we identify are pipelining and branch prediction.

2.4.1 Pipelining

When a sequence of instructions are independent of one another, they could be executed in parallel without affecting the outcome of that instruction sequence. This property called Instruction Level Parallelism (ILP) can be exploited in a number of manners, the simplest of which is to build is a pipelined processor [HP03].

Pipelining divides machine instructions into a number of stages. In order to be executed, an instruction must pass through each one of those stages. Because there is now hardware to support several stages in the execution of an instruction, there is no reason why several different instructions cannot be executing in the same CPU at the same time. The sole restriction is that each of the instructions must be at a different stage of execution. The effect of this is that several instructions can be executed in parallel, all at different stages in the pipeline.

There are 5 main stages in a classic processor pipeline:

1. Instruction fetch (IF)
2. Instruction decode/register fetch (ID),
3. Execute (EX),
4. Memory access (MEM),
5. Write back (WB) of results to the registers.

Figure 2.3 shows this type of classic 5 stage processor pipeline from [HP03]. In the first cycle, op_1 is at the first stage of the pipeline. In the second cycle, op_1 moves to the second stage and op_2 moves to the first stage. When all five stages of the pipeline are full, five different operations will be active, all at different stages of execution.

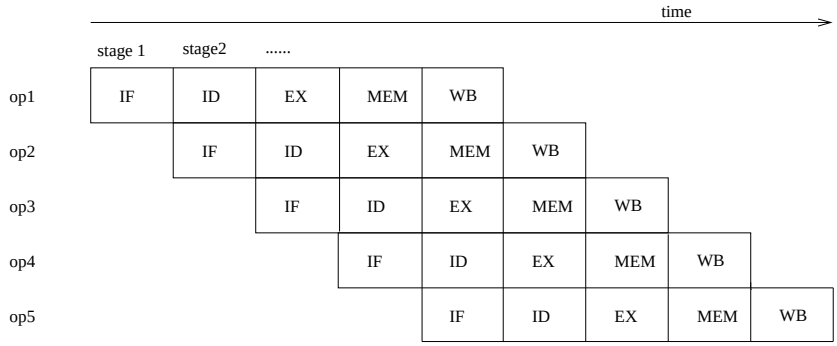


Figure 2.3: Classic processor pipeline

Pipelining requires a change in the way one considers the atomicity of the CPU clock. In a simple CPU without pipelining, a clock-tick would only need to be as short as the shortest instruction the CPU can execute. Once pipelining is introduced, the role of the clock changes to synchronising the pipelining, each stage of which is relatively short. Therefore pipelined CPUs tend to have faster clock rates. An interesting point of note is that while clock ticks must be uniform, the amount of time required by each stage of the pipeline can be different. In a sense, the pipeline can only be as fast as its slowest stage. A significant challenge in pipeline design is to ensure that the various stages are matched as closely as possible in terms of time required to do their job. A nice example of this effect in action can be found inside the Pentium 4 [HSU+01]. The pipeline in this CPU has two 'drive' stages which are solely for moving signals across the chip. If these signals had not been given their own stages and instead incorporated together or into another stage, the entire pipeline would have to be slowed down.

Some recent processors have even longer pipelines than the five-stage example in Figure 2.3. For example, Intel's P4 processor (Northwood core) has a 20 stage pipeline [ISS05], which is achieved by dividing the normal pipeline stages into smaller steps. Even more extreme is the Prescott core P4 processor with 31 pipeline stages [ISS05]. One advantage of a longer pipeline is that each stage takes a shorter amount of time, allowing the clock frequency of the processor to be increased.

In an ideal world, a pipeline of length n will have n instructions at various stages of execution, and complete one instruction per clock cycle. This rarely happens for a number of reasons, including dependencies among instructions and branch mispredictions (Section 2.4.2). At a certain stage, the amount of extra Instruction Level Parallelism (ILP) that can be exploited by extending the pipeline tails off. The exploitation of ILP through pipelining does not come for free. As the length of the pipeline increases, the synchronisation and the transfer of data from stage to stage also incurs more and more overhead. Therefore choosing the optimal length of a pipeline becomes a balancing act of trying to maximise ILP by extra pipelining stages, while trying to minimise the cost associated with those extra stages. In the next section we highlight a significant cost associated with longer pipelines.

2.4.2 Branch Prediction

We have deliberately deferred one significant source of pipeline stalls until this point, namely stalls due to control flow changes. While pipelines work extremely well for straight line code, once conditional branches start occurring, things become more difficult. In a pipelined processor when a conditional branch is read into the CPU's pipeline, the CPU does not know what the result of the condition will be, since the condition has not actually been evaluated yet. The branch may need to pass through several, even all, stages of the pipeline before the CPU knows where the next instruction should be fetched from. In order to keep the pipeline 'fed' with instructions, the CPU guesses where the next instructions after the conditional branch will come from, and starts feeding these instructions speculatively into the pipeline. This approach is called *speculative execution* and its primary purpose is to keep the pipeline busy [She04].

The problem with speculative execution is that it is only of value when the prediction is correct. In cases where the CPU makes the wrong guess about the conditional branch, it may have to clear out all the instructions that had been speculatively loaded into the CPU. Worse still, some of these instructions may have been partially executed, and any changes resulting from these partially executed instructions must be undone. The latter problem is fixed by adding hardware support into the CPU for speculative execution, to ensure that partial execution of an incorrectly guessed stream of instruc-

tions causes no problems. The former problem can only be fixed by guessing right all of the time.

Unfortunately, predicting such branches correctly 100% of the time is not possible. Furthermore, the problem of branch prediction is even more difficult than simply trying to predict conditional branches. In these cases, the task of prediction is to determine if a branch will jump or not. Indirect branches are much more complicated to predict. These are instructions which jump to a computed address, and thus can have multiple possible targets. The problem is exacerbated by longer pipelines, as a single misprediction by the speculative execution engine can result of the flushing of the entire pipeline, and a delay of n cycles (where n is the length of the pipeline) until instructions are being completed again. This is disastrous for instruction completion rates.

There are two reasons to be optimistic though. Firstly, not all applications contain lots of unpredictable branches. For example media and games applications contain a lot of predictable straight-line code with infrequent branches. Secondly, CPUs that have speculative execution also have branch prediction and branch target prediction algorithms and supporting hardware which enables better guesses to be made as to the control flow of the program. Typically, the longer the pipeline (and hence the bigger the branch misprediction penalty), the more sophisticated the prediction algorithms.

One of the simplest approaches to branch prediction is a *static branch predictor* [Pat95]. Static branch predictors use a set of static rules to determine whether a branch is taken or not. A simple static predictor might assume that a backwards branch is always taken. This approach would work well for loops but would fail spectacularly for other types of branches. Most modern branch target prediction algorithms are dynamic and work on the basis of a history. A common version of dynamic branch prediction uses a Branch History Table to guess whether a branch is taken or not. Simple 1-bit BHTs make guesses about whether a branch is taken based on what happened the last time the branch was encountered. Other BHT algorithms take more sophisticated approaches, attempting to exploit patterns in a branches history.

For dealing with the addresses of indirect branches, a common approach is to include hardware called a Branch Target Buffer (BTB) into the CPU [Fog01]. This BTB maintains a list of previously encountered branches (hashed by their address) and their previous target. If the branch is encountered again, a lookup to the BTB will find the entry for that branch, and speculative execution will continue at the indicated address.

If later, that address turns out to be wrong (i.e. if a branch misprediction occurs), the BTB entry for that branch will be updated with the correct address. BTBs are limited in size, though, and even in a program with numerous predictable branches, the BTB may not have space for all of them due to some of them being displaced out of the BTB by more recent entries. These type of BTB misses are called *capacity misses*.

The best performing indirect branch predictors at present are two level predictors. The first level is a global or at least partially shared set of history registers which store a limited number of previous targets of branches. This table of registers is called a Path History Register (PHR). The number of branches stored in the history at each entry is called the *path length*. The second level is a table called the Pattern History Table and is composed of 2-bit counters, each with an associated branch target address. It is this 2-bit counter that decides whether to replace the associated branch or not, when a misprediction occurs. First proposed by Driesen and Hölzle[DH98], these predictors have yet to be implemented on a wide scale.

Looking at some current desktop CPUs, the Pentium 4 [HSU+01] has a pipeline of 20 or 31 stages (depending on which CPU core one looks at). It has a BHT size of 4096 entries and a similarly sized BTB. The branch prediction algorithm is not public knowledge but appears to be quite efficient. The Pentium-M chip [GRA+03] has an undisclosed pipeline depth estimated between 10-20 cycles and has an extra piece of branch prediction hardware called a Loop predictor which is a specialised BHT specifically for loops. Interestingly, the Pentium-M chip appears to have a two level branch prediction unit that yields significant improvements in branch prediction accuracy [GRA+03].

2.5 Conclusion

In this chapter we have presented some of the background and history of the Java Virtual Machine. We have examined a number of types of execution engine, interpreter-based execution-engines being of greatest interest to us due to their utility in embedded systems. We have also examined some of the architectural features of a typical modern CPU which will be seen to have a substantial effect on JVM performance in subsequent chapters. In the next chapter we present a more detailed examination of the state of the art in JVM optimisation, paying close attention to previous research in the area.

Chapter 3

Literature Survey

3.1 Introduction

The Java Virtual Machine uses a stack-based bytecode to represent the program. Executing this bytecode is similar to executing normal machine code. The JVM fetches the next instruction, and based on the type of instruction and its operands, some action is performed. Thus the JVM must perform some action from a large number of alternatives, based on the value of the opcode. This is known as *dispatching* the bytecode instruction.

Instruction dispatch typically consumes most of the execution time in virtual machine interpreters. The reason is that most VM instructions require only a small amount of computation, such as adding two numbers or loading a number on the stack, and can be implemented in a few machine code instructions. In contrast, instruction dispatch can require up to 10-12 machine code instructions, and involves a time consuming indirect branch. For this reason, dispatch consumes a large proportion of the running time of most efficient interpreters [EG01].

Switch dispatch is the simplest and most widely used approach. The main loop of the interpreter consists of a large `switch` statement with one `case` for each opcode in the JVM instruction set. Ertl [Ert95] presents a number of 'C' source code samples and the corresponding compiled MIPS assembly language which we use in this chapter to illustrate the various threading techniques. Figure 3.1 shows how this approach is implemented in C, and Figure 3.2 shows the corresponding Mips assembly language.

```
typedef enum {
  add /* ... */
} Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  int *sp;

  while (1)
    switch (*ip++) {
    case iadd:
      sp[1]=sp[0]+sp[1];
      sp++;
      break;
    /* ... */
    }
}
```

Figure 3.1: Instruction dispatch using `switch` [Ert95]

```
$L2: #for (;;)
 lw    $3,0($6) #$6=instruction pointer
 #nop
 sltu  $2,$8,$3 #check upper bound
 bne   $2,$0,$L2
 addu  $6,$6,4  #branch delay slot
 sll   $2,$3,2  #multiply by 4
 addu  $2,$2,$7 #add switch table base ($L13)
 lw    $2,0($2)
 #nop
 j     $2
 #nop
 ...

$L13: #switch target table
 .word   $L12
 ...

$L12: #add:
 ...
 j   $L2
 #nop
```

Figure 3.2: Switch dispatch in MIPS assembly [Ert95]

Switch dispatch is simple to implement, but rather inefficient for a number of reasons. First, most compilers produce a range check to ensure that the opcode is within the range of valid values. In the JVM this is unnecessary, since the bytecode verifier already checks this. Secondly, the `break` is translated into an unconditional jump back to the start of the loop. Given that the loop already contains a jump, it would be better to structure the loop as a set of routines that jump to one another.

3.2 Dispatch Techniques

Historically, interpreters have not been designed with efficiency in mind. A survey of several interpreters by Romer *et al* [RLV+96] concludes that work to improve interpreters would be better spent at the software rather than hardware level. Ertl et al [EG01] examine efficient interpreters however and find that they contain a substantial

hardware-related inefficiency. This inefficiency relates to the unpredictability of branching from one VM to the next (instruction dispatch) and the fact that most modern CPUs are pipelined, some quite deeply. They found that 3.2%-13% of all executed instructions are indirect branches. This, in itself, is not a surprise, since an indirect branch will typically follow each VM instruction, while the VM instructions themselves are relatively simple operations, comprising a handful of machine instructions. The most surprising result they present is the number of cycles that these indirect branches consume, which is reported as 61%-79% of machine cycles. They highlight the importance of a good predictor, reporting a speedup of 2.55 or more over no predictor. From a software point of view, they recommend replacing a standard switch dispatch based scheme (Figure 3.3) with a direct threaded dispatch scheme.

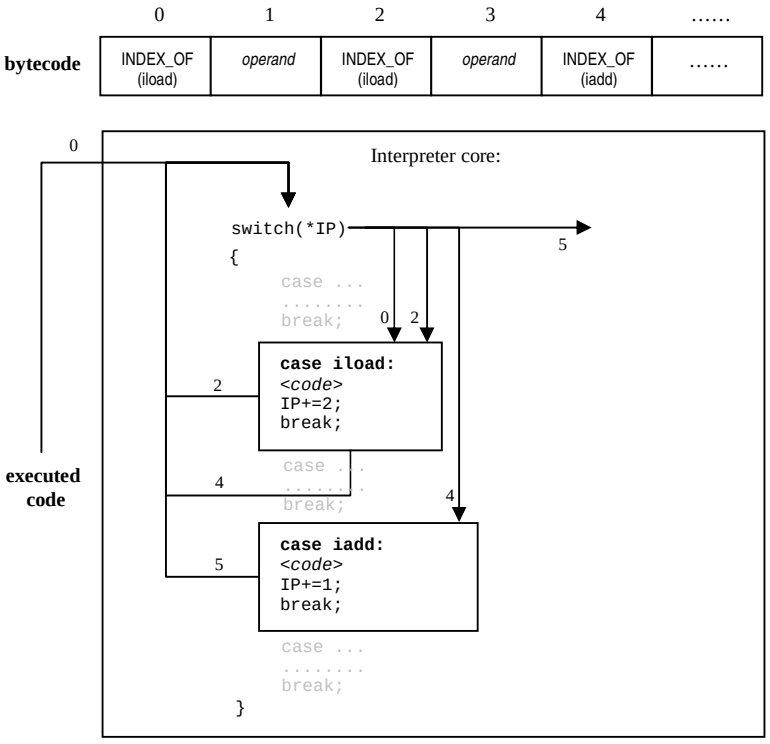


Figure 3.3: Switch Dispatch

3.2.1 Direct Threaded Dispatch

The first paper discussing threaded code was published in 1973 by James Bell [Bel73]. This technique was later rechristened direct threaded code in the light of many other variations on the same theme. The paper describes a process where the instructions in the code to be interpreted are represented by the address of their implementation, rather than a string, index or other types of representation. Instruction dispatch then becomes a process of retrieving the address of the next instruction from the instruction stream and jumping to this address. This optimisation is dependant upon the ability to treat labels as first class values (*first-class labels*). In cases where this support is not present, one can either resort to a more portable dispatch method, or indeed resort to writing sequences of machine code for the dispatch part of the interpreter instructions. First class labels are not part of the ANSI-C standard. However, `GCC`, the popular multi-platform GNU compiler has support for first-class labels. This feature allows the addresses of labels to be treated like any other pointer. Figure 3.4 shows an example of a threaded dispatch interpreter.

The advantages of direct threaded dispatch over switch dispatch are twofold. Firstly direct threaded dispatch involves only a load and an indirect branch as opposed to the slightly more complicated switch instruction. The switch instruction indexes a table of branch targets which usually includes incurring the cost of a bounds check in addition to a load and indirect branch. Typically a direct threaded dispatch needs three to four machine instructions, whereas the switch dispatch needs nine to ten machine instructions [Ert94]. Secondly, and more importantly, because each VM instruction gets its own dispatch code in the direct threaded approach, it means that there will be more entries in the BTB. This has a massive contribution to the predictability of branches. Instead of having a single indirect branch, branching to all possible instructions, a larger number of branch points exist, each branching to a subset of possible VM instructions[1]. Consider the loop shown in Figure 3.6. The Figure shows the state of the BTB after a complete iteration of the loop (including the branch backwards).

[1]This subset can be an effective subset resulting from the sequences of instructions seen in a program. It can also be an absolute subset where the interpreter rules dictate that only certain instructions can follow a particular instruction. For example in Java, if the `iload` instruction is encountered (pushing an integer to the operand stack) one is guaranteed that the next instruction is not `fadd` (add two floats on the top of the instruction stack).

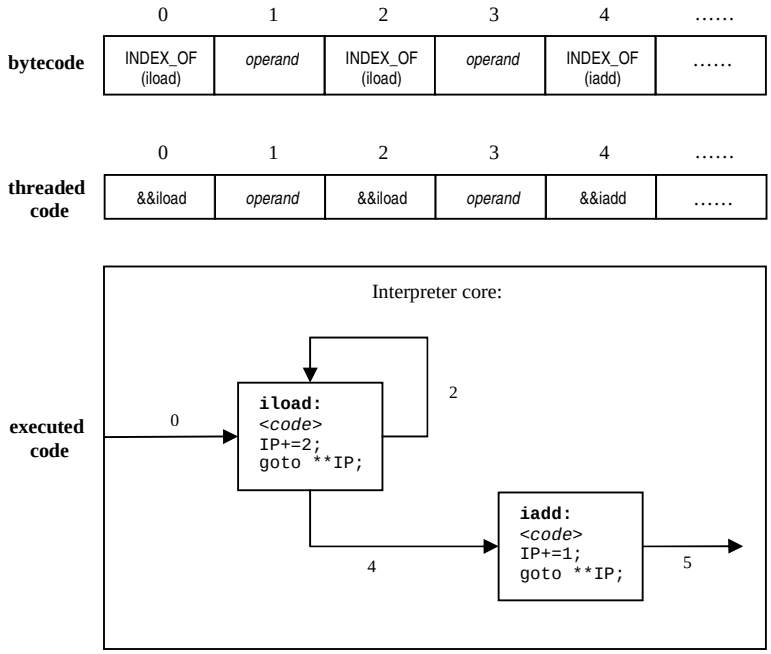


Figure 3.4: Threaded Dispatch

```
typedef void *Inst;

void engine()
{
  static void * program[] = { &&iadd /* ... */ };
  Inst *ip;
  int *sp;

  goto *ip++;

 iadd:
  sp[1]=sp[0]+sp[1];
  sp++;
  goto *ip++;
}
```

Figure 3.5: Instruction dispatch using direct threading in GNU C [Ert95]

In a switch based dispatch scheme, each instruction dispatch will most likely cause a branch misprediction. This is because no two copies of the same instruction occur in succession and the BTB entry for the dispatch in the switch statement will never guess correctly. However in the direct threaded approach because `iload` always follows `iadd`, `isub` always follows `iload` etc. in the loop, each of the individual dispatches at the end of a VM instruction is always to the same target. Overall Ertl et al [EG01] estimate that threaded dispatch gives an increase in branch prediction accuracy of 2%-20% from switch based dispatch to about 45% with threaded dispatch.

In work describing the efficient SableVM interpreter, Gagnon et al [GH01] describe the use of direct threading in their interpreter. Although they provide performance comparisons between SableVM and othe VMs, they do not report any results that are meaningful in terms of determining how much direct threading contributes to SableVM's performance. In later work Gagnon et al measure the speedup of a threaded version of their interpreter over an atypically optimised[2] switch-based version of their interpreter. They measure the effect on running time using the SPECjvm98 benchmarks [SPE98] and two object oriented applications soot [VRCG+99] and SableCC [GH98], and obtain an average speedup of 1.12.

Gregg et al [GEK01] also describe the construction of an efficient Java interpreter using direct threading dispatch. Although results are presented comparing the interpreter's performance in relation to a number of other interpreters, no results are presented that measure the exact contribution of direct threading to that performance.

3.2.2 Token Threaded Dispatch

One valid criticism of direct threading is that it involves replacing instruction codes in the instruction stream with the addresses of the instructions. Apart from the necessary code translation, this also can cause code bloat in the interpreted code (*bytecode bloat*), depending on the interpreter's code representation and the size of a memory address[3].

[2]The switch-based interpreter they used is actually switch-threaded. It uses a switch statement but does not operate on bytecode, but on optimised word-sized code. Therefore the speedup obtained by using direct threaded dispatch should be treated as a lower bound that one might obtain over a normal switch-based interpreter operating on bytecode.

[3]For example in Java on a 32-bit x86 CPU, a single bytecode representing an instruction must be replaced with 4 bytes (a single 32-bit address).

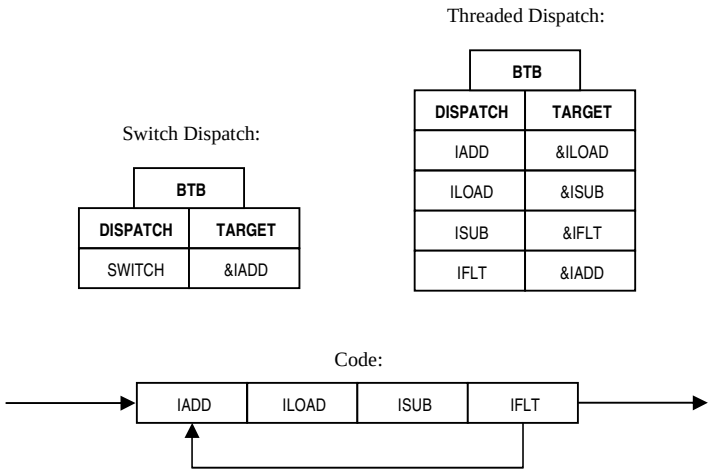


Figure 3.6: Switch Versus Threaded Dispatch

An alternative dispatch mechanism, *token threading* is shown in Figure 3.7. This dispatch scheme avoids any modifications to the interpreter's instruction stream. A table of instruction addresses is used and, while each VM instruction still gets its own dispatch code, the dispatch itself is changed. Each dispatch consists of using the standard instruction code, found in the instruction stream, to lookup the address of that instruction in the table. Once the address of the instruction has been loaded, an indirect branch takes place to that address. Because token threading is identical to direct threading, apart from the table lookup, it gives identical performance in terms of branch prediction accuracy. Despite the table lookup, token threading could actually improve performance due to the fact that it does not cause code bloat, and therefore ought to give better cache performance.

3.2.3 Offset Threading

If one is concerned about the table lookup, another alternative is to use *offset threading*. This approach attempts to address the bytecode bloat issue, but some instruction stream translation is still required to replace instruction codes. This time however, a base address is selected and shorter offsets to each instruction are stored into the instruction stream instead of full memory addresses. For example, instead of storing a

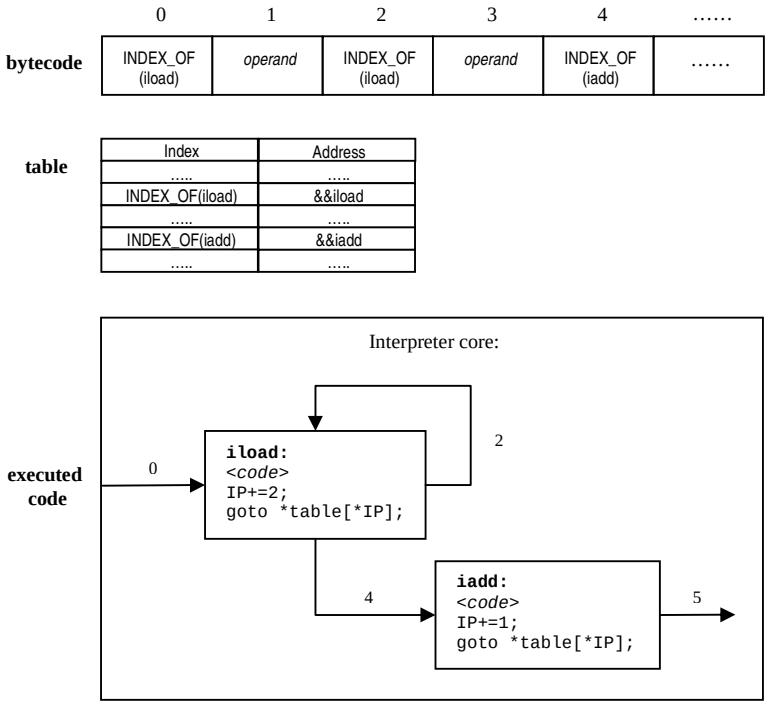


Figure 3.7: Token Threaded Dispatch

```
typedef void *Inst;

void engine()
{
  static Bytecode program[] = { iadd /* ... */ };

  Bytecode *ip;
  Inst dispatch_table = { &&nop, &&aload_null, .... };
  int *sp;

  goto dispatch_table[*ip++];

 iadd:
  sp[1]=sp[0]+sp[1];
  sp++;
  goto dispatch_table[*ip++];
}
```

Figure 3.8: Instruction dispatch using token threading in GNU C [Ert95]

```
lw   $2,0($4) #get next inst., $4=inst.ptr.
addu $4,$4,4  #advance instruction pointer
j    $2       #execute next instruction
#nop          #branch delay slot
```

Figure 3.9: Direct threading in MIPS assembly [Ert95]

```
lhu  $2,0($4)  #get next inst., $4=inst.ptr.
addu $4,$4,4   #advance instruction pointer
addu $2,$2,$5  #add base pointer
j    $2        #execute next instruction
#nop           #branch delay slot
```

Figure 3.10: Offset threading in MIPS assembly [Ert95]

32-bit address A into the instruction stream, one could store a 16-bit offset B into the instruction stream, where $A = Base + B$. Instead of a table lookup before the indirect dispatch, a cheaper addition can take place. This cost of this addition can be cheaper if one uses a variation of offset threading called *segment threading*. In this approach, the segment register addressing mode of the host CPU is used.

The segment register is loaded with the code segment containing the VM instruction core and offsets are stored in the instruction scheme as before. Because the addition is absorbed into the addressing mechanism, it can be marginally faster than offset threading. Unfortunately this type of addressing is only available on x86 CPU derivatives and even if it is available, may require assembly code to ensure it is used. Additionally the offset requires 16-bit alignment.

The offset threading scheme was introduced by Barnhart [Bar83] where he proposed a scheme for implementing a direct threaded interpreter on the 32 bit Motorola 68000 using only 16 bit addresses. The scheme allocates one register to be the "base pointer", which contains the starting address of the interpreter. All other addresses are offsets relative to the base pointer. Thus, we need a 32 or 64 bit base address, and provided that the interpreter code fits inside 64K, the threaded code can consist of a series of offsets to that base address.

3.2.4 Indirect Threaded Dispatch

One way to reduce the size of direct threaded code is to factor our frequently occurring immediate constants in the code. Typically, the same constants appear many times in the code and the goal of *indirect threaded* [Dew75] dispatch is to have a special version of each instruction for each immediate constant value that the instruction can have. This is achieved by adding an extra level of indirection. Instead of the code consisting

```
lw   $2,0($4) #get the next inst, $4=inst.ptr.
#nop          #load delay slot
lw   $3,0($2) #get the VM instruction address
addu $4,$4,4  #advance instruction pointer
j    $3       #execute next instruction
#nop          #branch delay slot
```

Figure 3.11: Indirect threading in MIPS assembly

of a list of addresses of executable routines interspersed with immediate operands, the code becomes a list of pointers to `struct`s. Each `struct` contains field with a pointer to the routine to implement the instruction, and another field containing the immediate operand.

Indirected threaded dispatch reduces the code size by factoring out multiple copies of frequently used constants. However, the addition of a struct for each VM instruction-operand pair offsets some of the savings in memory size. Furthermore, the extra level of indirection requires an additional load to the compiled dispatch code when compared with direct threading (see figure 3.11).

Some implementations of the Forth language, such as GForth [Ert93], use a hybrid of direct and indirect threading. As with indirect threading a `struct` is created at translation time for each combination of VM instruction and operand in the program. However instead of the first field containing a pointer to the executable code, it contains a machine language instruction to jump to this code. This combines many of the benefits of direct and indirect threaded dispatch. Unfortunately, this scheme requires that executable machine code be generated at run time, something that cannot be done portably. Furthermore many architectures with separate instruction and data caches, such as the Intel Pentium, impose a very heavy performance penalty if there are cache lines containing both data and executable instructions. For these reasons, we do not consider this scheme any further.

3.2.5 Other Threading Mechanisms

Call threading [Ert96] is threading mechanism which is viable for most compilers, even those not supporting first-class labels. Instead it relies on indirect calls instead of

indirect jumps. Each dispatch involves a call and a return sequence. In addition any shared data between VM instructions such as the stack pointer and instruction pointers must be declared as global variables. Apart from the overhead of calls and returns, without radical changes this approach is unsuitable for VMs that use multiple, concurrent, OS-level threads. This is because a context switch between threads would require the overhead of switching the shared data stored in global variables. With OS-level native threads, this might not even be possible, since the JVM might not know a context switch had occurred. Ertl [Ert96] provides measurements on the number of cycles required for a dispatch on two processors, the R3000 and R4000, for switch, direct and call threading. While call threading is not as efficient as direct threading, it is a bit more efficient than switch-based threading.

Another variant is *bit threading*, a technique specific to the NOVIX NC4000. A single bit in a 16-bit opcode determines if the opcode is a call to routine or an executable instruction. This scheme allows a single-cycle call to a subroutine. A similar scheme is used for returns. These processor-specific techniques are not portable and require assembly language to implement them. We do not consider them further.

Subroutine threading [Kog82] is yet another variation on direct threaded code. First presented by Curley [Cur93a, Cur93b] for Forth on the 68000, in this approach the instruction stream is no longer interpreted. Instead it is composed of a sequence of machine code `call` and `ret`(urn) instructions to/from various subroutines representing VM instructions. A recent variation of subroutine threading termed *context threading* has been developed by Berndl et al [BVZB05]. Each VM instruction is implemented with a C function. Instead of interpreting bytecode or threaded code, a very simple just-in-time compiler generates executable code for a sequence of calls to these functions. This eliminates indirect branches completely from the dispatch of VM instructions, at the cost of some loss in simplicity and portability.

3.3 Instruction Scheduling

One option to improve Instruction Level Parallelism and hence performance on deeply pipelined architecture is to reorder instructions so that values are not used immediately after they are computed. Avoiding this means avoiding pipeline stalls, which are expensive on pipelined architectures. This technique of re-ordering instructions is termed

instruction scheduling [CMC+91, CMW+94]. Typically performed within basic blocks, the re-ordering is constrained by data dependencies, the violation of which implies the changing of the semantics of the code. In short, not all orderings of instructions are possible, and it is the job of the instruction scheduler to satisfy these constraints while maximising ILP.

Software pipelining, a specific case of instruction scheduling attempts to overcome memory latency in loops [RG81, Lam88]. It does this by overlapping pre-fetches for a future iteration of the loop with the current iteration. This is something of a balancing act, as on one hand fetches must be early to avoid memory latency, but on the other hand, if they are too early they may get flushed from the cache before they are actually used. Because of the irreducibility of the interpreter loop [ASU86], software pipelining must be performed manually. One such technique is to load the address of the next dispatch as early as possible, even moving the load into the previous instruction [EG03b] (but may involve having an additional move instruction). This results in a pipeline of two stages. Figure 3.12 shows a three stage pipeline optimisation (in-

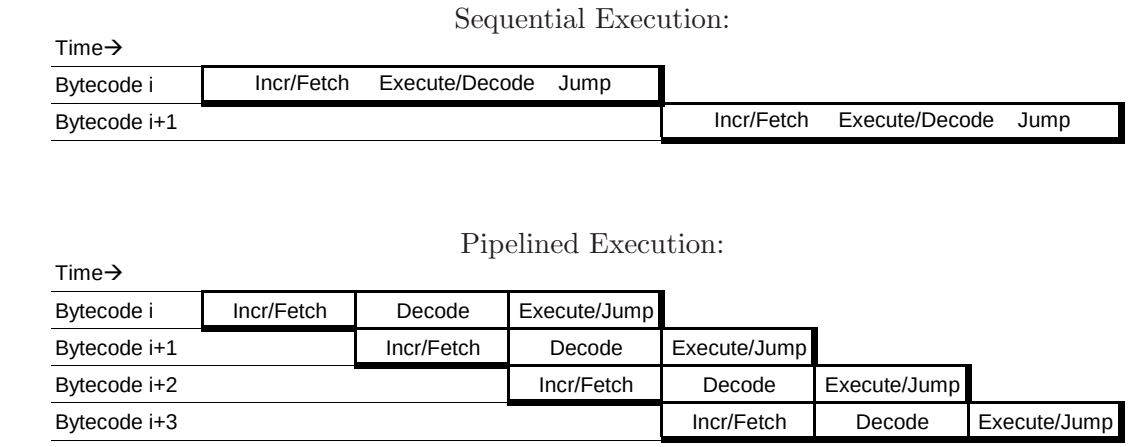


Figure 3.12: Sequential and Pipelined Interpreters

crement/fetch, decode and execute/jump) which was applied by Hoogerbrugge et al [HATW99] on the Philips Trimedia VLIW processor [SRD96]. Although they do not isolate the performance benefits of the hand-coded software pipelining alone, they do report an improvement from an average of 6.27 cycles per instruction without software

pipelining and stack caching to 4 cycles per instruction with both optimisations. In subsequent work Hoogerbrugge et al [HA00] also report execution speed improvements of 19.4% and a cycle reduction of 14.4% from pipelining their threaded, stack caching JVM. Smaller optimisations for Prolog are also presented by Costa [Cos99].

These techniques can be of great assistance on pipelined architectures with delayed indirect branches, a prepare-to-branch instruction or some equivalent. This includes processors such as the Philips Trimedia and the Motorola PowerPC. On architectures where branches cannot be resolved early, such as x86 CPUs, branches are predicted in hardware and the programmer cannot control this prediction process directly. Even in such processors, instruction scheduling can offer marginal performance improvements by scheduling the load for the address of an indirect branch a couple of instructions before the actual indirect branch. By the time the actual dispatch reaches the execution stage, the load will have resolved. However a branch misprediction may still occur, depending on what prediction the branch prediction unit has made for that indirect branch. On such processors, this optimisation reduces the latency associated with fetching these loads, but does not significantly contribute to reducing the frequency of cost branch mispredictions.

Work exploiting ILP in Java has been examined in the context of just-in-time compilers where the scheduling problem is more tractable due to the additional knowledge of bytecode sequences at runtime. Suggestions for annotating bytecode allowing an intermediate code compiler share information allowing better run-time instruction scheduling (and other optimisations) is presented by Reig [Rei01]. The effect of run-time exceptions on the ILP of JIT generated native code is examined by Arnold et al [AHKR00]. Other work has concentrated on architectural models for exploiting ILP. Watanabe et al [WCL01] present an architecture for exploiting ILP and Thread Level Parallelism for Java. These results are promising as they present simulations that show their architecture can achieve 7.33 Effective Instructions Per Cycle (EIPC) with 8 slots and a 4 instruction scheduling window for each slot.

3.4 Stack Caching

Most operations in the Java VM access the operand stack, whether popping or/and pushing a value. In order to speed up these operations, it would be advantageous

to employ some form of *stack caching*, keeping the stack contents in registers. Most processors however have low numbers of registers available for the programmer to use directly so keeping the entire stack in registers is simply not practical.

Fortunately, various studies have shown that the depth of the Java operand stack is quite shallow [PWL04, Por04]. This implies that only a few registers would be required to store a significant portion of the stack. Even if one were to just cache the topmost item of the stack, this would give result in a stack-cache hit rate of over 50% [PWL04].

There are three classes of stack caching mechanism, differing in implementation details and functionality:

1. Fixed-size stack caching.
2. Dynamic stack caching.
3. Static

3.4.1 Fixed-size Stack Caching

This approach to stack caching assigns a certain number of registers and an ordering among those registers to hold items at the top of the stack. For example, if three registers, R_1, R_2, R_3 are assigned as a stack cache, then one register, say R_3 must be designated as the top of the stack at all times. Another, say R_2 must be designated as the second from top, and another as the third from top.

To support this approach, the VM must be redesigned. Firstly VM instructions need to be rewritten so that stack accesses are to the registers. Secondly, code for the maintenance of the stack cache must be introduced. If for example three items are stored in the cache and a push occurs, then R_3 must be flushed to memory, R_2 must be shifted to R_3, R_1 to R_2 and then the push can take place to R_1. Figure 3.13 shows an example of the shifting of registers required when a push occurs when the stack cache is full. Note that, apart from the register-register copies, there is only one store to memory.

Similar complications can arise when an item is popped, as values need to be shifted among registers, and this time a value must be read in from memory into R_3. Apart from this housekeeping, a stack caching Java interpreter must also flush its state to memory at appropriate times, for example before a garbage collection occurs, or before

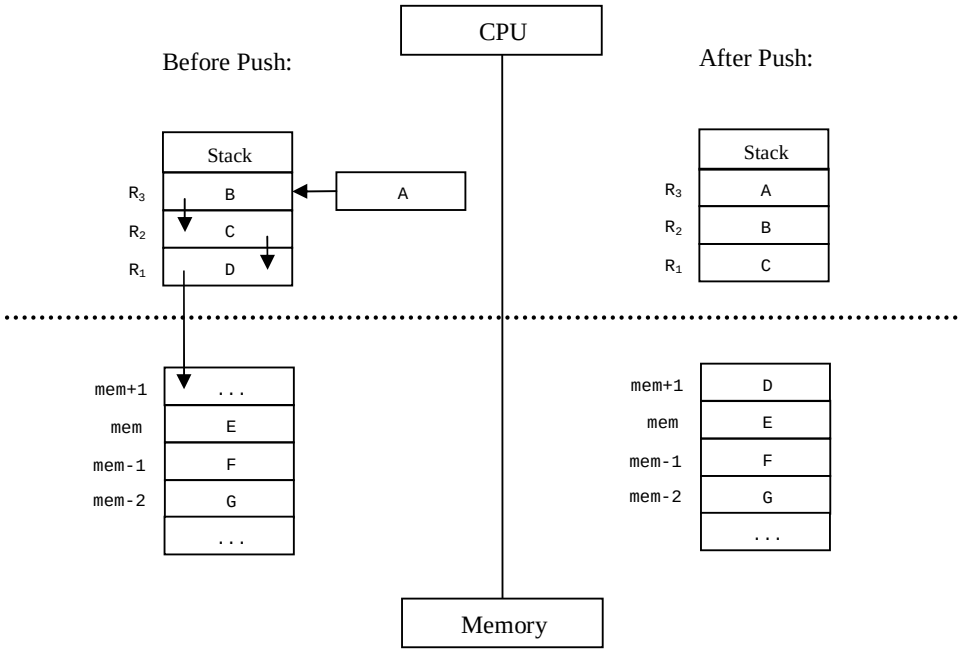


Figure 3.13: Fixed Size Stack Caching with Three Registers

certain VM instructions. The major overheads of this form of stack caching are the register-register moves which take place each time an instruction has a net effect on the stack size. Other stack-size maintaining instructions benefit enormously from this approach. For example the type conversion instructions in Java such as `getfield` (which pops one object reference off the stack and pushes another) will find its operand in a register and will push back to that register when done, all without register-register copying.

Although there are still loads or stores each time a net change to the stack takes place, the important feature is that these loads and stores do not represent data read or written to/from the current VM instruction. The total number of loads and stores executed may not be reduced. But when performing an operation such as an integer add, the operands are immediately available in registers, rather than needing to be fetched from memory. Thus the critical data dependence path in the code is shorter. Values may need to be loaded, but these loads can be scheduled concurently with the add, rather than the add having to wait for these loads to complete. Thus, the loads

are moved off the critical data dependence path. Hoogerbrugge et al [HATW99] use a fixed size cache of two registers in a pipelined interpreter on the Philips Trimedia VLIW processor (see Section 3.3 for results).

One configuration of fixed-size stack caching is much more practical than others. This is to have a fixed-size cache of only one register. When the stack height changes, no shifting of values between registers is needed, although the usual loads and stores are required. This is the aproach used in the GForth interpreter. Ertl [Ert95] performs an empirical comparison across a range of benchmarks in Forth. In this work it was found that keeping a fixed number of items cached was inefficient except for the topmost item. This is due to unnecessary loads and stores needed to maintain the constant size. In the absence of a full implementation, results are presented in terms of eliminated stores, loads and stack pointer updates.

Peng et al report an implementation of Xorp, a hybrid stack JVM on Intel's XScale processor in [PWL04]. In their implementation, they employed a fixed size cache of two items. This choice was based on the fact that there are only seven callee-saved registers, three of which were already required by the JVM (for the instruction pointer, stack pointer and local variables pointer). Of the four that remained, one was required for a code sharing (between stack-caching states) mechanism, and another was left free in order for `GCC` to generate efficient code. According to their tests on selected J2ME benchmarks, 80% of executed VM instructions read and write exclusively from/to the stack. The novel contribution of this paper is their mechanism for sharing code between interpreter states, preventing the code-explosion problem associated with stack caching. Their interpreter compares favourably to an un-cached threaded implementation, giving a 13.6% average speedup on their selected benchmarks.

3.4.2 Dynamic Stack Caching

The fixed-size stack cache approach, although relatively straightforward to implement, has the overhead of a large number of register-register copies. For VLIW machines, this may be an acceptable cost given the instruction scheduling opportunities that arise. However, for non-VLIW, machines the benefits are not as great and therefore the relative cost of these register-register copies is higher. An alternative approach that avoids the register-register copies, is to allow the number of cached items in the stack

cache to vary. This means that a load or a store is no longer inevitable when a stack-height changing VM instruction occurs. Loads will only occur when a VM instruction cannot find the correct number of items in the stack-cache to pop (the stack cache is empty), or when there is not enough room in the stack cache to push a result (the stack cache is full).

Implementing such a variable-sized stack is a little more complicated, since the top of stack can be any of the stack-cache registers (or none if the stack is empty). To solve the problem, the concept of states is introduced. For example, if the interpreter has a stack-cache of three registers R_1, R_2, R_3, then the interpreter can be in one of four states:

State 0 The stack is empty.

State 1 The stack-cache has one cached value. R_1 is the top item.

State 2 The stack-cache has two cached values. R_2 is the top item.

State 3 The stack-cache is full. R_3 is the top item.

At any time the interpreter is in one of these states. In order to maintain the state in an efficient manner, four sets of VM instructions (i.e. four interpreter cores) must be defined, one for each possible state. For example, consider the `iload` instruction which pushes an integer onto the stack. In **State 0** there would be a $\texttt{iload}_0$ instruction that writes an integer to R_1 and dispatches to the next instruction. However, it isn't that simple any more. Since there are now four copies of each VM instruction, it must dispatch to the **State 1** copy of the next VM instruction, since there is now one item in the stack-cache. For example if the next instruction to be executed is also `iload`, then a dispatch will take place to $\texttt{iload}_1$, the **State 1** copy of `iload`. Similarly, when $\texttt{iload}_1$ pushes to the stack, it writes to R_2 and when finishes, dispatches to a **State 2** copy of the next instruction. Also when the **State 2** instruction $\texttt{iload}_2$ pushes to the stack, it writes to R_3 and when finishes, dispatches to a **State 3** copy of the next instruction. The **State 3** instruction $\texttt{iload}_3$ is a little more interesting because the stack is full in this state. One solution is to flush an item out of the bottom of the stack (R_1) into the in-memory stack. Then the other items in the stack (R_2,R_3) can be shifted down. Finally the instruction can push the integer value onto the stack (i.e.

store it in R_3). Because the stack was full, and is still full, there is no state change. The `iload`$_3$ instruction will dispatch to the **State 3** copy of the next instruction. Ertl [Ert95] discusses a number of other strategies for the case where the stack cache becomes full.

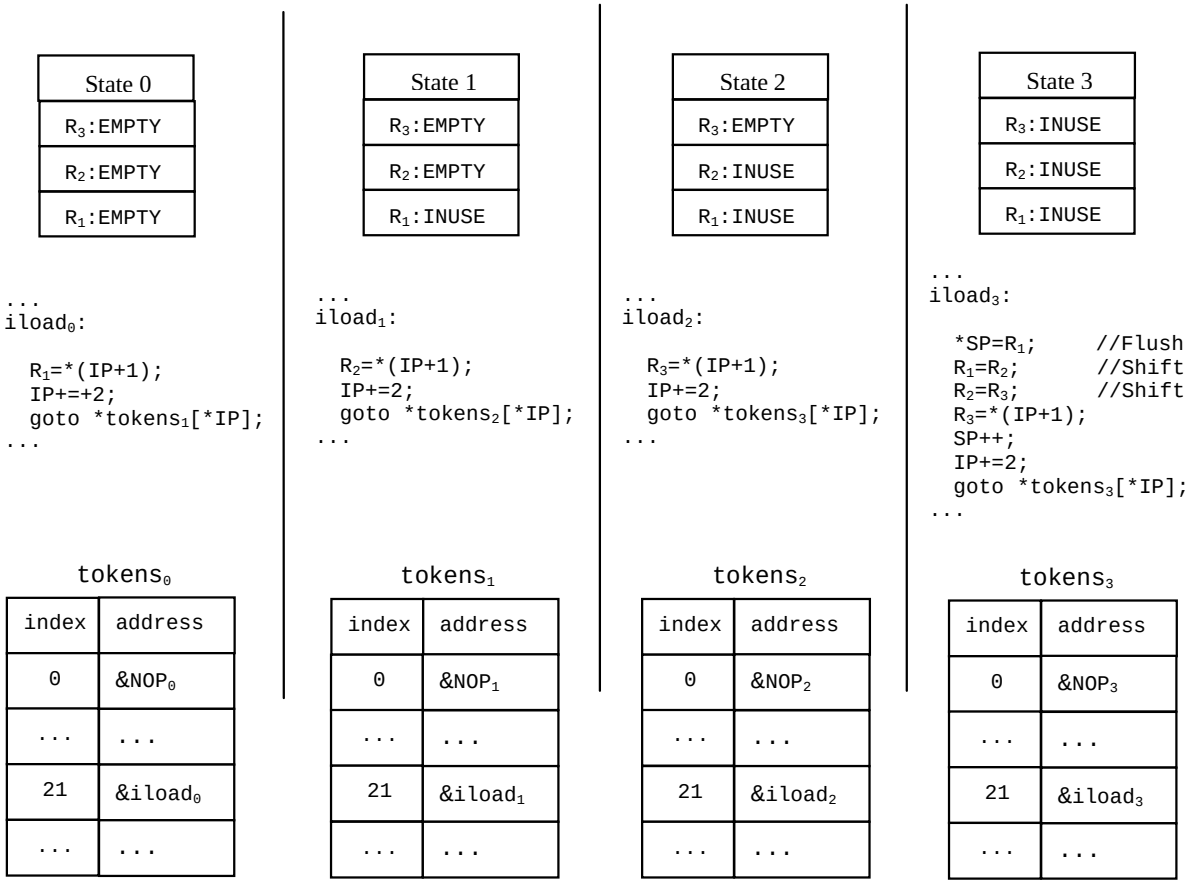


Figure 3.14: Dynamic Stack Caching with Four States

Figure 3.14 shows the various implementations of the `iload` instruction, one for each state. Note how switching from one state to another is implemented, namely by using a table of dispatch target addresses. This is a variation of token threading, but we no longer branch to an entry in a single array, *tokens*, to execute the code for instruction i. Instead, there are now four arrays of tokens, and we must also specify the state j we want to enter and branch to $tokens_j[i]$. Because of this dependency on the tables of

dispatch target addresses, this type of stack-caching requires token-threaded dispatch. Another point of note in the example is the omission of stack pointer updates in all cases except where an item is being pushed out of the stack into memory. Although it isn't shown in the example, a stack pointer update will be performed when more items are popped off the stack than are in the cache. For example, in **State 0** when a pop instruction occurs, the in-memory stack must be used and therefore the stack pointer must be modified.

Dynamic stack caching has been used on hardware stack machines [Bla77, HS85, HFWZ87, HL89, Koo89] and for improving procedure call performance on the Bell-Labs Machine Project [DM82] and also at UC Berkeley [HP03, Fur88]. The first interpreter work in this area was reported by Debaere et al [DV90] on Forth an Modula-2 VMs. The experimental evaluation, unfortunately is limited to the sieve benchmark. The speedups they report[4] are quoted as 16% for Forth on an 8086, using a 2-register cache and a speedup of 17% for the Modula-2 VM on a 68020, using a three register cache.

There are two problems with this stack-caching mechanism. As noted above, dynamic stack caching is dependant on token threaded dispatch. This precludes us from using the faster direct threaded dispatch. An additional problem is one of interpreter size. For a cache-size of n items there is a total of $n + 1$ states, and for each state a copy of the interpreter core. This is referred to as the *code-explosion* problem.

In relation to Java, the HotSpot VM uses dynamic stack caching with one register. So the stack cache can be empty, or can have one item cached [Gri99]. The HotSpot VM has a separate state for each type that can be at the top of the stack. Because only a limited set of instructions can operate on a particular type, each state in the interpreter does not need to implement the full JVM instruction set. For example, when the item at the top of the stack is an float, the JVM will be in an float-caching state. The `iadd` instruction does not need to be implemented in this state, since it cannot operate on a float.

[4]It isn't quite clear from their paper what these speedups represent. It appears that the speedups are intended to be from VMs with no stack items cached in registers.

3.4.3 Static Stack Caching

An alternative static stack caching mechanism *static stack caching* is presented by Ertl [Ert95]. This approach removes the need for the interpreter to track the state of the cache by transferring the bulk of the work to the compiler. The different instruction cores, one for each state are still required as before, however. The main technique of

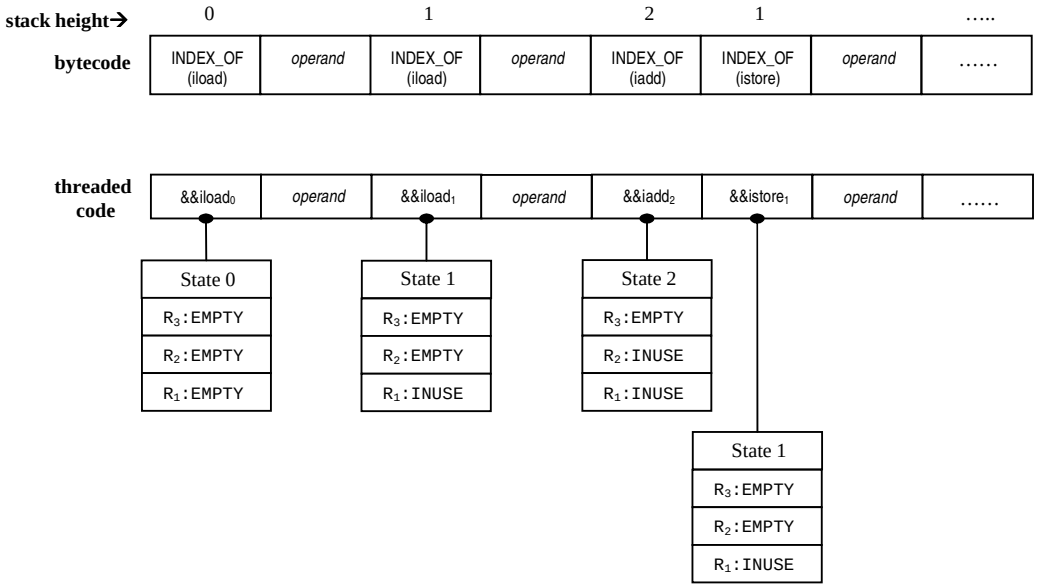


Figure 3.15: Static Stack Caching with Three Registers

static stack caching is shown in Figure 3.15 where the code sequence `iload_2 iload_1 iadd istore` is shown before static stack caching and after. The stack height before each instruction is shown above the instruction, and each threaded instruction is tagged with a diagram showing the current stack-cache state. Note how the translated code contains the stateful instruction sequence $\texttt{iload\_2}_0$ $\texttt{iload\_1}_1$ $\texttt{iadd}_2$ $\texttt{istore}_1$. Although the code in the example is threaded code, this stack caching mechanism will also work with token threaded dispatch.

This approach is possible, largely in part due to the strict rules Java has about stack height. The height of a stack must be zero upon method entry and exit. In addition, the stack height must be fixed at all points in the opcode. For example, if the height of the stack is h immediately before an execution of an instruction in the bytecode,

then it will always be h at that point in the bytecode, even a backwards branch occurs and previous instructions get executed again. This static knowledge of stack height allows the compiler to determine which interpreter core to use for an instruction at a particular point in the bytecode, based on the knowledge of the stack height for that point. One positive side effect of such a scheme is that `pop` instructions can be optimised away completely. Assuming a stack height of 3 on entry to the sequence of bytecodes `iadd pop iload`, the static stack algorithm can convert this to either $\texttt{iadd}_3\ \texttt{pop}_2\ \texttt{iload}_1$ or simply $\texttt{iadd\_1}_3\ \texttt{iload}_1$. In addition to this optimisation, Ertl also reduces stack pointer updates which no longer need to be performed after each VM instruction.

While the code-explosion problem is still present, it can be controlled by not providing for less frequent instruction-cache-state combinations. The compiler can avoid entering these combinations by inserting additional stack manipulation code in the rare case where the combination would normally be entered.

The previously published work detailed in this section points toward stack caching as an effective optimisation, but there are two problems associated with the approach. Firstly, it relies on a ready availability of registers. However, on register starved processors such as x86 CPUs, there is unlikely to be enough registers to cache a significant number of stack-items. Even if one could acquire enough registers for a stack cache, the code generated by the compiler would degrade due to a likely increase in the number of register spills. Secondly, while several efforts above have addressed the code explosion problem, it is still present and may cause serious problems with increased branch mispredictions. In particular, we feel that the code-sharing approach presented by Peng et al [PWL04], while clearly effective at reducing code-explosion, will cause increased numbers of branch mispredictions. This opinion is based on the fact that the optimisation increases the number of branch targets substantially in relation to the number of dispatch points.

3.5 Instruction Specialisation

Although Java's use of a stack architecture reduces the number of operands per instruction, a significant number of operands remain. In order to reduce some of the loads associated with these operands and to contribute to more compact code, specialised

versions of many instructions in Java have become part of the standard instruction set [GJSB00]. The loads and stores are particular targets for specialisation, based on the presumption that certain combinations of instruction/operand combinations will occur frequently. For example, the `iload` instruction copies an integer from a local variable onto the stack. It takes one argument, namely the number of the local variable to be copied from. The `iload_0` instruction, on the other hand takes no arguments, loading an integer from local variable 0 onto the stack. While specialised instructions are no doubt an important addition to the Java VM instruction set, they are by no means vital, since a generic version can always be used in the place of a specialised instruction (as long as the appropriate operand is introduced into the bytecode scheme). Indeed, some specialisations (such as `fstore_0`) are used so rarely that we cannot justify their inclusion in the instruction set [DHPW01]. This raises the possibility of creating new specialisations based on bytecode analysis. These new specialisations can be introduced into the bytecode in a just-in-time manner, while the lesser-used specialisations can be stripped out of the bytecode at the same time.

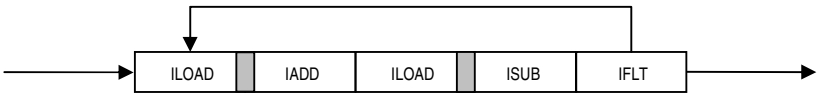
Such specialisation techniques have not been evaluated fully in Java before. Vengupal et al [VMK02] proposed optimising Java interpreters for embedded systems using semantically enriched code (sEc). The idea of sEc is to profile the application and generate specialised instructions specially for that application. Unfortunately no implementation was ever created and therefore no results are available. In contrast, an implementation of the SICStus Prolog virtual machine with specialisation is presented along with results in Nässén [NÖ1] and Nässén et al [NCS01]. While they experience code space savings in the order of 8%-16%, instruction specialisation does "not yield any speedup except in a few cases, contrary to expectations". They report results for both the i686 Celeron and SUN UltraSPARC architectures and note that specialisations give better performance on the i686, most likely due to its register starved nature. This optimisation can alleviate some of that pressure. Specialised instructions tend to compile, at least in part, to real machine instructions with immediate operands. This frees up registers for other purposes, a behaviour which will have a greater effect on register-starved architectures.

3.6 Static Replication

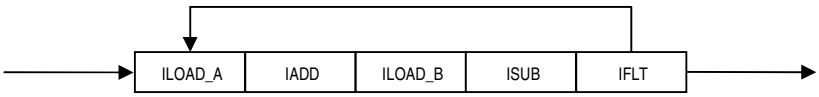
A novel technique presented by Ertl et al [EG03a] for improving branch prediction on machines with BTBs is static replication. The concept is to create copies of commonly VM instructions at interpreter compile-time. At runtime the bytecode of the interpreter is modified so that the interpreter will use the copies in a round robin fashion. At a high level, one can view this optimisation as making more use of the BTB, due to the fact that it now has more entries, the original indirect branches plus the additional replicated ones. At a low level the optimisation can be viewed as modifying the bytecode so that each VM instruction is only likely to occur once in the working set, and therefore branch mispredictions are less likely. Figure 3.16 illustrates this effect for a simple loop where two replicas of the `iload` instruction, `iload_0` and `iload_1` are available. Before replicas have been laid down, the loop in the example will cause two branch mispredictions per iteration. After replicas have been laid down in a round-robin fashion, there are no longer any branch mispredictions per iteration of the loop.

To choose instructions for static replication, the authors profile running code and select the most frequently executed VM instructions. The authors report a speedup of up to 2.39 on GForth using this technique with 400 replications. This approach is completely portable, but if there is to be any benefit must be used in combination with a threading mechanism that moves the dispatches into the VM instructions (Section 3.2). Static replication on a switch-based interpreter would actually make performance worse. The authors also note that they tried an alternative random placement for replicas in the bytecode, instead of the round robin scheme. This proved to be an inferior approach, due to the fact that random placement did not have the same spatial separation between replicas, and thus the same VM instruction was more likely to occur in a working set, for example a loop. If random placement was used in Figure 3.4 to choose possible replicas for the `iload` instruction, both `iload`s could get replaced by the same replica. This is just as bad as the case without replication, causing two branch mispredictions per iteration of the loop.

Before Replication:



Round Robin:



Random (possible):

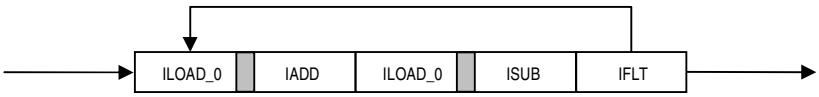


Figure 3.16: Adding static replications using Round Robin and Random placement.

3.7 Superinstructions

In a study on the structure and performance of interpreters, Romer et al [RLV+96] observe that Perl and Tcl interpreters are high-level interpreters; they execute thousands of machine instructions per interpreted instruction. MIPSI [Sir], a MIPS interpreter and the JVM are both low-level interpreters; they execute fewer than 100 (and for most VM instructions, less than 10) real machine instructions per VM instruction on average.

This implies a higher dispatch to real work ratio in the JVM interpreter. One possible way to address this problem is to change the instruction set of the JVM, introducing *superinstructions* which are larger instructions composed of commonly occurring sequences of bytecodes. For example, in the JVM, the instruction sequence `aload getfield` commonly occurs in the bytecode. It is possible to create a new instruction `aload_getfield` which does the work of both separate instructions. The bytecode can then be modified (possibly in a just-in-time manner) to ensure the new superinstruction gets used.

Superinstructions have been used for many years but the first cited work in this area relates to superoperators, introduced by Hughes [Hug82]. These superoperators are an optimisation based on λ-calculus expressions for functional languages. Proebstring [Pro95] introduces superoperators using `hti`, a token threaded hybrid translator/interpreter for ANSI 'C'. The interpreter runs intermediate bytecode stored in `lcc`'s intermediate representation [FH95]. The interpreter's instructions are the nodes in the Intermediate Representation tree (IR-tree). The author records that `hti` runs a factor of 8-16 times slower than native code. By introducing superoperators, which coalesce many atomic operations into a single operation, the performance of `hti` is boosted by a factor of 2-3 times. However, to achieve this level of performance, superoperators must be customised to suit a particular program. This means that the program to be run must be available at the time that the interpreter is built. Similar performance will not be reached on other programs. Proebstring also details that the heuristics chosen for selecting superoperators gave a 78%-81% reduction in code size.

In later work, Wegdam [Weg96] improved the selection heuristics which were apparently causing compilation times of several days. As the emphasis of this work was on compact code generation, no execution times were presented. However a code size

reduction of 50%-70% is reported. With a similar purpose in mind, Hoogerbrugge et al [HATW99] use superinstructions on a decision tree intermediate format compiled from C/C++ code. They report that with only 150 superinstructions, code size can be reduced by 30%. They observe that adding too many superinstructions can increase the size of the interpreter and thus harm execution speed.

Nässén [NÖ1] and Nässén et al [NCS01] present some work they carried out on superinstructions[5] for the SICStus VM. Because the intent of their work was to specialise interpreters for particular benchmarks, the selection of candidate superinstructions was based on the profiling data for these benchmarks. They report that superinstructions gives an average reduction in code size of about 9% and a reduction in execution time of 8-10%. They also examine the effects of specialising superinstructions for commonly occurring operands, but conclude that this approach has little positive effects beyond those achieved by superinstructions alone.

The effect of superinstructions on branch prediction rates are examined by Ertl et al [EG03a] in GForth [Ert93]. In this paper they identify and present results of an additional benefit of using static superinstructions, namely that they expose larger code windows to the compiler. Given two VM instructions A, B, with sizes $|A|$ and $|B|$ respectively, an efficient compiler should generate code for the superinstruction AB with a size $|AB| < |A| + |B|$

They use the profiling and static superinstruction support built into their interpreter tool vmgen [EGKP02] to extend the GForth instruction set. They use a dynamic selection method, selecting the most commonly executed sequences of instructions in a single benchmark. Then, using these superinstructions they evaluate the performance of the new VM on benchmarks that the VM has not been specifically profiled for. Their results indicate that superinstructions are good at reducing executed machine instructions, but their greatest benefit is reducing branch mispredictions.

The problem of selecting an optimal set of superinstructions for a particular program is NP-Hard [Pro95]. This problem and the task of parsing the bytecode for superinstructions are essentially the same problems that arise during dictionary-based text compression [BCW90], specifically selection of an optimal dictionary, and then application of the compression itself. Ertl et al [EG03a] note that there are two possible algorithms for parsing bytecode; greedy and optimal, although they only investigate

[5]The author(s) use the term *instruction merging* rather than superinstructions.

the former. Greedy parsing is easier to implement, involving applying the longest matching superinstruction at each point in the bytecode. Optimal parsing is a little more difficult, but fortunately the problem can be solved efficiently using dynamic programming [Bel73].

A number of superinstruction selection algorithms are examined for Forth by Gregg et al [GEW01]. They compared a number of selection heuristics basing their performance on the number of dispatches eliminated. They found that selection based on static occurrences gave substantially better performance than a dynamic one. They conclude that the dynamic measure is not entirely suitable for selecting a universally performing VM due to the tendency of loops in profiled code to skew superinstruction frequencies in a way that was only likely to favour the code being profiled.

A Java specific analysis of the problem is examined by O'Donoghue et al [OP04] where they analyse basic block profiling information from a modified version of the JamVM [Lou03] running the CaffeineMark benchmark suite [Pen97]. They choose superinstructions on the basis of dynamic occurrences but average them over several programs, avoiding bias towards any particular program. Applying an iterative selection, they select the most common sequence and re-adjust the frequencies of the remaining sequences and repeat until they have selected the required number of superinstructions. Being limited to implementing the superinstruction codes in bytecode, they only had a budget of 10 superinstructions to add to the JVM. However, this still resulted in a speed improvement of 18% in a switch based version and 14% in a threaded version of the interpreter.

In extensive work, Eller [Ell05] examines various issues regarding superinstructions in Forth and Java, in particular the performance of optimal versus greedy parsing algorithms and concludes that greedy parsing achieves near optimal results. Also in the same work, Eller examines a number of superinstruction selection heuristics and concludes when the number of superinstructions permitted is larger than about a thousand, the best heuristic is simply to construct superinstructions from all possible subsequences up to length 4.

Stack based bytecode	Register based bytecode
`iload1`	`move r10,r1`
`iload2`	`move r11,r2`
`iadd`	`iadd r10,r10,r11`
`istore3`	`move r3,r10`

Table 3.1: Conversion of bytecode to register code (with stack pointer=10)

3.8 Register Machines

There are two competing views as to the best architecture register or stack within which to implement a VM. While register hardware has prevailed over stack based hardware in more recent CPUs, stack based architectures still remain predominant for virtual machines and is the chosen architecture for the two most currently prominent VMs, the Java VM [GJSB00] and the Common Language Runtime, Microsoft's commercial implementation of the Common Language Interface (CLI) standard [ECM02]. This follows a long tradition of stack based virtual machines such as Pascal-P [Nel79], Modula-2 [Woo93] and Forth[RCM96]. Typically the arguments for such an architecture center around the fact that operands for VM instructions are implicit; they will always be the items at the top of the stack. The main arguments stemming from this characteristic of stack machines are that:

1. Compilers for stack based VMs are simpler than compilers for register based VMs since they do not need to perform register allocation.

2. Stack VM code is more compact since operands and destination results are implicitly consumed from and stored to the top of the operand stack.

More recently, interest in register based architectures has increased. Lua, the embeddable scripting langauge [IdC05] is implemented on a register based VM, and the next version of Perl (Perl 6) will run on a register based Parrot VM [RST04]. Despite a continuing discussion over the years as to which architecture is superior [Mye77, SM77, MB99, WP97] no quantitative comparison had been presented until recently.

Davis et al [DBC+03] present a working system for translating stack based JVM code to register based code. Table 3.1 illustrates a typical example of the type of

```
Before copy propagation:
                        move r10,r1       //iload1
                        move r11,r2       //iload2
                        iadd r10,r10,r11  //iadd
                        move r3,r10       //istore3
```

```
After copy propagation:
                        iadd r10,r1,r2    //iload1,iload2,iadd
                        move r3,r10       //istore3
```

Figure 3.17: Example of forward copy propagation

conversion they carry out. In the example the stack pointer is assumed be at 10 and to push positively. Note the code growth as a result of the initial transformation.

They examine and discuss a number of design issues such as method handling and copy propagation within or across basic blocks. It is this copy propagation procedure which is used quite successfully to tackle the code growth issue in the translation from stack to register based code. Figure 3.1 illustrates an example of applying the copy propagation algorithm to the code generated in Table 3.1.

Empirical studies were carried out using the SPECjvm98 [SPE98] and Java Grande [BSW+99] benchmark suites. Overall they found that translating to a register based format decreased the number of executed VM instructions by 34.88% while increasing the number of loads by 44.81%. In addition they found only a small reduction (1%) in eliminated instructions when moving from copy propagation inside basic blocks to copy propagation across basic blocks. Given the relative slow speed of copy propagation across basic blocks, they recommend the former approach. With respect to the increased loads, they point out the the average increase of 2.32 loads per dispatch eliminated is a promising figure, given the high cost on deeply pipelined CPUs of the branch mispredictions that result from dispatches.

Shi et al [SGBE05] built on the work presented by Davis et al [DBC+03] by building an actual register machine JVM and making several improvements to the translation from stack based to register based code. The actual translation takes place in a just-in-time manner, although the authors note that this is not may not be the ideal approach

```
Before copy propagation:
                        iadd r10,r1,r2    //iload1,iload2,iadd
                        move r3,r10       //istore3

After copy propagation:
                        iadd r3,r1,r2     //iload1,iload2,iadd,istore3
```

Figure 3.18: Example of backward copy propagation

in the long term[6]. Backward copy propagation (and a second phase of forward copy propagation) along with moving constant instructions out of loops are some of the improvements they make. Figure 3.18 illustrates the effect of applying backward copy propagation to the previously forward copy propagated code from Figure 3.17.

The new translation mechanism reduced the number of executed VM instructions by 47.21%. They also report the number of additional loads at being approximately 1.07 times of the number of dispatches eliminated. As mentioned above, the cost of these additional loads is negligible compared to the cost of the dispatches which no longer needs paying. Runtime performance on the register based JVM they constructed improved by approximately 30% over a corresponding stack based JVM. They measure performance improvements on switch-based and threaded dispatch interpreters and report a slightly better performance improvement for the switch based register JVM over the switch based stack JVM (30.69% versus 29.36%). This is due to the additional cost of dispatches (many of which are removed in a register architecture) in a switch based interpreter.

While these recent results make a compelling argument in favour of register based VMs, other work in the area attempts to address the criticisms levelled at stack-based architectures. In particular VanDrunen et al [VHP01] present a scheme for replacing loads and stores of local variables with stack manipulation instructions, a form of stack allocation (as opposed to register allocation). Although they do not provide an implementation, they report results on some transformations of a number of selected benchmarks. The results point to a reduction of 2% to 25% of loads and stores as a result of their optimisation. Other work in the area has been carried out by Maierhofer

[6]Although they estimate that the translation consumes less than 1% of execution time.

et al [ME98] and Koopman [Koo92]. We recall with interest the argument presented at the beginning of this section; namely that compilers for stack based VMs are simpler than compilers for register based VMs since they do not need to perform register allocation. While this may be true, recent work suggests that compilers for stack based languages ought to be doing stack scheduling instead.

3.9 Dynamic Code Copying Techniques

Although VM instruction set enhancements such as superinstructions, specialisations and replication give good speedups, all these optimisations suffer from the same problem. Essentially they try to optimise the interpreter for all possible programs, based on the profiling data of a limited set of programs. For any new program, the set of instruction enhancements is hardly likely to be optimal. A dynamic VM instruction optimisation, on the other hand, could respond at runtime by examining the bytecodes of the program to be run, and modify the interpreter according to those bytecodes, thereby tailoring the JVM to that particular program. Just-in-time compilers [DS84] are an extreme form of this in action, translating bytecode to native code at runtime.

Using `memcpy` to construct new sequences of executable code from shorter sequences of executable code at runtime may seem unrealistic. However, this is the approach taken successfully by Piumarta et al [PR98]. In this work they present *selective inlining*, a dynamic method of instruction optimisation based on code copying using `malloc` and `memcpy`. To implement this technique, they start out with a threaded interpreter. They divide up VM instructions into two sets, relocatable and non-relocatable, depending on whether the implementation for the VM instruction can be copied to a new area of memory and retain the same semantics. For example, a VM instruction containing a call to a function relative to the program counter cannot be relocated to a new area of memory since this would involve a changed program counter. They then create a *no-copying list* to identify those instructions that cannot be relocated.

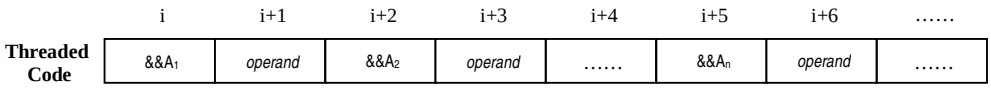
At runtime, they scan through the threaded code and try to find straight-line sequences of relocatable instructions, A_1,A_2,...,A_n. When they find such a sequence, they reserve a new area of memory address, $ADDR$, for inlined code. Then for each instruction in sequence in the bytecode, they copy the implementing executable code for that instruction from the VM interpreter core into the new area of memory, concatenating

the VM instruction implementations together in a new area of memory. Since the sequence of instructions A_1,A_2,...,A_n is a straight-line sequence, the dispatches at the end of the instructions do not need to be copied. Therefore, after the copying routine, the new area of memory will contain the concatenated code for a superinstruction (with no dispatches). A single dispatch must still be added to the end of the copied sequence of executable code however.

To ensure the new dynamically created superinstruction is used, the threaded code sequence where A_1,A_2,...,A_n is located must be modified. As this sequence in the threaded code is actually a sequence of addresses (perhaps with intervening operands) it will actually read &&A_1,&&A_2,...,&&A_n. This is modified to $ADDR$,&&A_2,...,&&A_n. This will force a jump to the newly created superinstruction if this threaded code is encountered during program execution. As with a regular superinstruction, this new instruction maintains the interpreter's program counter, so when the dispatch at the end of the superinstruction is encountered, it will dispatch to whatever instruction occurs after the sequence &&A_1,&&A_2,...,&&A_n. Figure 3.19 shows this process, before and after the inlining process. The work presented here has two interesting behaviours with respect to space. Firstly Piumarta et al do not attempt to compress the threaded code by removing the redundant pointers &&A_2,..., &&A_n. This would cause complications with instruction operand offsets and instruction pointer increments. Secondly, to prevent the same dynamic superinstruction being created (replicated) several times, they use a hash table into which they enter superinstructions that are created dynamically. When a new sequence of instructions that are suitable for superinstruction creation is encountered, the hash table is consulted before superinstruction creation starts. If the instruction sequence is found in the hash table, then the pre-existing superinstruction is used. Therefore the same dynamic superinstruction might 'cover' a number of threaded code sequences. Piumarta et al evaluate their technique using both a fine-grained RISC-like interpreter and the coarser-grained Objective-Caml interpreter [Ler97] over a number of benchmarks.

Even for the previously optimised Objective-Caml interpreter, the authors report a minimum speed improvement over the non-inlined interpreter of 50% across the Objective-Caml benchmark suite [Ler96]. The results for the finer-grained RISC-like interpreter gave an average speedup across all tested architectures and benchmarks of 1.68. The authors conclude by noting that, as this technique is a dispatch elimination

Before Inlining:



After Inlining:

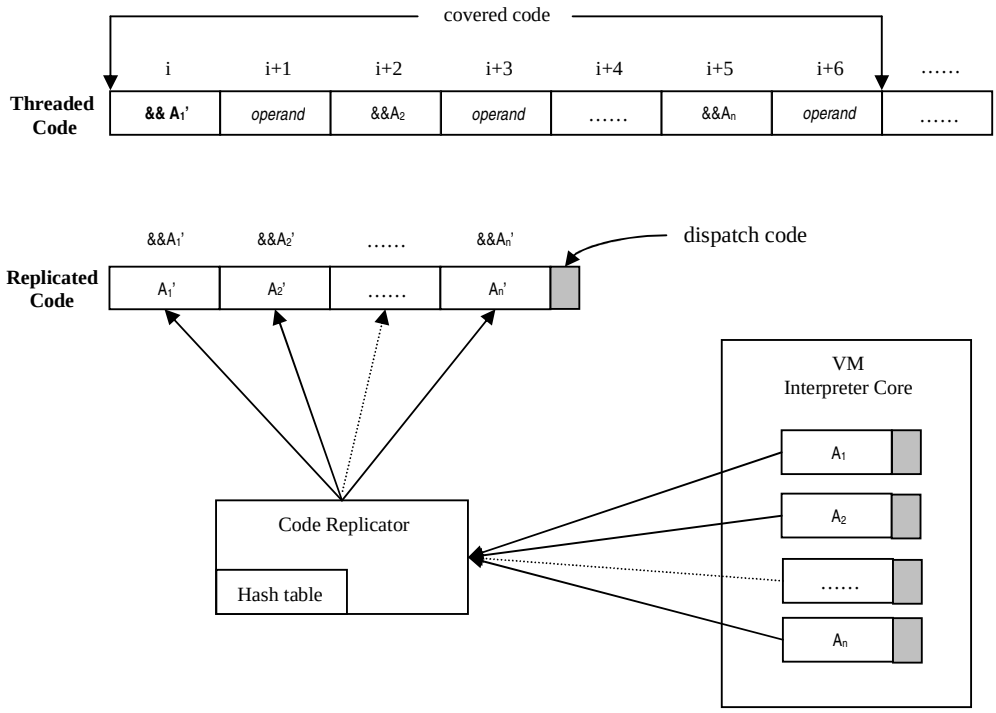


Figure 3.19: Dynamic Superinstructions (inlining) with threaded code.

technique, it will be of greater benefit to finer-grained interpreters with a high dispatch to work rate.

In a description of the SableVM, Gagnon et al [GH01] note that they provide an option for using Piumarta's code copying technique for creating basic block sized dynamic superinstructions. In this work they do not present the details of how this was implemented, nor of speedups from the technique[7]. In later work, Gagnon et al [GH03] present *preparation sequences*, a method for supporting dynamic superinstructions in Java. This technique is an efficient way to overcome some of the difficulties associated with quickable instructions (Section 5.4.4) and possible race conditions resulting from multiple threads of execution. The technique itself does not change the type of superinstruction that is created; they remain no more than one basic block in length.

They measure the effect of dynamic superinstructions on running time using the SPECjvm98 benchmarks [SPE98] and two object oriented applications soot [VRCG+99] and SableCC [GH98]. This speedup is given as an average of 1.39 over a direct threaded version of SableVM without superinstructions.

The type of dynamic superinstruction introduced by Gagnon et al [GH03] is subtly different to that created by Piumarta [PR98]. Piumarta's technique re-uses dynamic superinstructions when a new sequence of instructions is found to match a previously inlined sequence. We use the term *dynamic superinstructions without replication* for this approach. In contrast, Gagnon et al create a new dynamic superinstruction each time an inlineable sequence of bytecodes is encountered, potentially creating multiple copies of the same dynamic superinstruction. We use the term *dynamic superinstructions with replication* for this approach.

3.10 Conclusion

In this chapter we have seen many optimisations for interpreters, ranging from different threading techniques to VM instruction enhancement and dynamic executable code copying. In particular we have seen how dispatch times dominate the execution times for interpreters and many of these optimisations attempt to reduce this effect. The optimisations of greatest interest to us are those that will yield the greatest performance

[7]It appears by implication from their later work that they did not attempt to inline quickable instructions, instead treating them as non-relocatable instructions.

across a number of architectures.

In the next chapter we present the Tiger interpreter generation tool which we will use to apply some of these optimisations in order to generate a fast, optimised JVM interpreter. This tool will allow us to generate a large portion of the code for the interpreter automatically and thus permit more powerful research options later in this thesis.

Chapter 4

Tiger - An Interpreter Generator

4.1 Introduction

Many of the optimisations discussed in this dissertation required extensive coding modifications to the CVM interpreter core. Therefore, a tool to automate these changes is of great value to a researcher working in the area. This tool could accept a description of an interpreter core in some domain-specific language and emit the code for the actual interpreter core, subject to the value of certain flags or parameters. These flags or parameters can affect what sort of code is generated, turning on or off various optimisations or even using particular variations of those optimisations. In addition, debugging code can be inserted by the interpreter generator. Again, this can be turned on or off as required. Similarly, other code such as profiling code can be injected into the generated code whenever requested by the interpreter tool user. One such tool, vmgen [EGKP02], allows the interpreter writer/researcher far greater power to include variations and combinations of optimisations without extensive hand-modification of the interpreter core.

In the initial stages of the project where we implemented threaded code and indirect threaded code variants of the interpreter, vmgen was extremely useful. Additionally, when profiling for simple superinstructions and including them in a new interpreter core, vmgen was more than sufficient. However, we identified a number of features that made vmgen a less than optimal choice. They were:

1. No built-in support for replication (static or dynamic).

2. No support for instruction specialisation.

3. Limited to sets of superinstructions where all prefixes are present.[1]

4. Greedy parsing only for superinstructions.

5. No support for replication of superinstructions.

6. No support for superinstructions across basic blocks.

In order to overcome these limitations we re-implemented vmgen in a way that ensured it was easy to maintain. Java was selected as the implementation language (vmgen was implemented in Forth) to encourage as wide a user-base as possible. In this chapter we discuss some of the functionality of the Tiger tool and also discuss how some of this functionality is implemented.

4.2 Tiger - Front-end Functionality

Tiger[2], the **T**rinity **I**nterpreter **GE**nerato**R**, takes a similar approach to vmgen in that, the user supplies a file describing the interpreter core and the tool then compiles this into a series of 'C' source files that can be included at the appropriate point in the interpreter source code. The input language was changed considerably both to allow for the extensions that were to be made, but also to increase clarity.

4.2.1 Definitions and Options

The Tiger source file begins with a number of definitions. Figure 4.1 lists some of the more commonly encountered definitions. The data stack is marked by the DATA keyword which is followed by the stack pointer, the type and the direction (increment when a pop occurs). The instruction stream is marked by the INST keyword which

[1]For example, in vmgen, if one wishes to define a new superinstruction composed of $instr_1$ $instr_2$ $instr_3$ $instr_4$ in that order, one also needs to define two other superinstructions, one composed of $instr_2$ $instr_3$ $instr_4$ and another composed of $instr_3$ $instr_4$.

[2]The Tiger tool was presented at CC2005 [CGE05b]. The tool documented in that paper was inspired by Ertl's vmgen[EGKP02], and designed, implemented and applied to a Java VM and a small 'C' VM by the author of this thesis.

```
DATA SP StackVal32   1
INST IP Address      -1
SIZE Uint32          4 SPEC
SIZE JavaVal32       4 SPEC
SIZE ClassBlock*     4
SIZE MethodBlock*    4
......
OPTION dispatchMethod token
OPTION earlyLoad off
OPTION stackUpdateCombining agressive
OPTION preferredOpcodes off
OPTION debugger off
OPTION profiling off
OPTION histogram off
```

Figure 4.1: Some definition and options in Tiger

is followed by the instruction stream pointer, the instruction stream type and the direction (increment when an item is read).

A list of data type sizes follows the instruction stream and stack definitions. This list contains all the types which are permitted to be stored in the instruction stream or the stack. For each new type that is added, two pieces of information are recorded. The first is the number of slots on the instruction stream or stack that the type occupies and the second is whether operands of the type can be specialised or not (see Section 4.2.8). The SPEC keyword indicates that it can, whereas absence of the keyword indicates that it cannot.

A selection of options are then available to set, all of which use the OPTION keyword. This keyword is followed by the option to be set and the desired value for that option. Options exists for various behaviours such as the injection of profiling code into the generated code and for many optimisations of the generated code.

4.2.2 General Syntax

A typical opcode defined in Tiger is depicted in Figure 4.2. The first token is the opcode name, then followed either by the stack behaviour (SP) or the instruction

```
ADD     SP( Int32 src1, Int32 src2 - Int32 dest ) //Stack behaviour
        IP( - next)                               //Stream behaviour
        dest=src1+src2;                           //User Code
---------------------------------                 //Separator
```

Figure 4.2: A typical opcode definition in Tiger

stream behaviour (IP). The stack behaviour specifies what types and instances needs to be popped off the stack before the core of the opcode is to be executed and what is to be pushed onto the stack after the core of the instruction has completed. The '-' symbol represents the separator between what is to be popped and what is to be pushed in the stack descriptor. The instruction stream behaviour allows us to specify which operands are to be loaded from the instruction stream (none in this case). The '-' symbol represents the end of the current instruction. The keyword next indicates that another instruction will follow in the instruction stream. The absence of the next keyword indicates that an unconditional control flow change will occur in the instruction. Tiger uses the stack and instruction stream descriptors supplied, along with the code core specified, to generate 'C' code for the instruction.

Ordering of behaviours: The order of popping and pushing to/from the stack and the reading from the instruction stream is well-defined in Tiger. The order is:

1. Pop stack items
2. Read from instruction stream
3. After executing instruction, push to stack.

This ordering is important for clarity, because the same variable name can appear in the stack popping/pushing and instruction stream behaviours. It can be useful because we can sometimes put a lot of the work of an opcode into the stack and stream behaviours, increasing the clarity of the code. For example, where a constant is being loaded from the instruction stream onto the stack, we can take an approach similar to that of Figure 4.3.

```
Tiger Code:     PUSHC SP( - Int32 c)
                      IP( Int32 c - next );
                ----------------------------------------
```

```
Generated Code: PUSHC:
                {
                        Int32 c;
                        vm_Int32_equals_Int32(c,IPPTR[1]);
                        vm_Int32_equals_Int32(SPPTR[0],c);
                        IPPTR=IPPTR+2;
                        SPPTR=SPPTR-1;
                        goto **IPPTR;
                }
```

Figure 4.3: Reading a constant from the instruction stream onto the stack

4.2.3 Dispatch Method

There are three dispatch mechanisms supported in Tiger through the use of the dispatchMethod option. This option can be set by the user to be one of:

switch This option creates a switch-based interpreter core. Each instruction begins with a 'case' statement and ends with a 'break;'. The number associated with the case is the index of the instruction, using the INDEX_OF(*instr*) macro (Section 4.3.3). Thus, it is assumed that the switch guard variable holds an instruction index.

token This option creates a token-based interpreter core. Each instruction *instr* begins with a label. The address of this label is stored in the Tiger-generated instruction address array VM_LABELS (Section 4.3.4) at index INDEX_OF(*instr*). It is assumed that the instruction indices are stored in the instruction stream IP, so a dispatch to an instruction involves getting the index from the instruction stream (IP[0] or *IP) and dispatching to the address stored at that index in the instruction address array (goto *VM_LABELS[*index*]).

direct This option creates a direct threaded interpreter core. This time it is assumed that the instruction address is stored in the instruction stream IP. A dispatch in

this case involves de-referencing the instruction stream and then dispatching to this address (goto **IP). The VM_LABELS array, used for the token threading option above, can also be quite useful in this dispatch scheme at translation time when the indices in the instruction stream are replaced by the corresponding instruction addresses.

Figure 4.4 shows the different versions of the ADD instruction generated by each of the options switch, token and direct.

4.2.4 Pushing of Expressions

To eliminate unnecessary variable declaration and to keep the Tiger source and generated code compact and readable, Tiger supports the pushing of expressions onto the stack. The key to this is the +NODEC modifier. Typically, Tiger will declare all items it sees in the stack and stream behaviours (variables for both reading and writing values). The +NODEC modifier prevents this from happening. Then we can write instructions such as in Figure 4.5.

Tiger will still push 'a+b' onto the stack, but it will not try to declare 'a+b'. The Int32 type associated with the expression is still important to have, since Tiger is supposed to track stack pointer updates, and therefore must know what type (and hence size) has been pushed onto the stack.

4.2.5 Eliminating Unnecessary Stack Writes

The vmgen tool provided a mechanism for eliminating unnecessary stack writes in special circumstances [EGKP02]. This is useful, for example when generating code that duplicates the topmost item on the stack (Figure 4.6). Normally this would involve popping the topmost item off the stack and then pushing that item back onto the stack twice. The first push is, strictly speaking, redundant since the item is already at that position in the stack. Therefore, the first item doesn't really require writing to the stack. All that is required is to modify the stack pointer as if the item had been written to the stack and then to proceed to write the second item as normal.

vmgen provides for this optimisation through the use of a special flag that can be turned on or off, as required. When this flag is set, vmgen will ignore a push of a

switch based:

```
case INDEX_OF(ADD):
{
    Int32 b;
    Int32 a;
    Int32 c;
    vm_Int32_equals_Int32(b,SP[0]);
    vm_Int32_equals_Int32(a,SP[1]);
    {
        c=a+b;
    }
    vm_Int32_equals_Int32(SP[1],c);
    SP+=(1);
    IP+=(1);
    break;
}
```

token threaded:

```
ADD:
{
    Int32 b;
    Int32 a;
    Int32 c;
    vm_Int32_equals_Int32(b,SP[0]);
    vm_Int32_equals_Int32(a,SP[1]);
    {
        c=a+b;
    }
    vm_Int32_equals_Int32(SP[1],c);
    SP+=(1);
    IP+=(1);
    goto *VM_LABELS[IP[0]];
}
```

direct threaded:

```
ADD:
{
    Int32 b;
    Int32 a;
    Int32 c;
    vm_Int32_equals_Int32(b,SP[0]);
    vm_Int32_equals_Int32(a,SP[1]);
    {
        c=a+b;
    }
    vm_Int32_equals_Int32(SP[1],c);
    SP+=(1);
    IP+=(1);
    goto *(IP[0]);
}
```

Figure 4.4: Different dispatch methods in Tiger

```
ADD     SP( Int32 a, Int32 b - +NODEC Int32 a+b )
        IP( - next);
----------------------------------------
```

Figure 4.5: Pushing an expression onto the stack

```
DUP     SP( Int32 a - Int32 a, Int32 a )
        IP( - next);
----------------------------------------
```

Figure 4.6: A candidate instruction for stack push elimination

variable of the same name, where that variable has been popped from the same position previously in the instruction.

A slightly different approach is recommended in Tiger[3]. Support for this optimisation had already been introduced for other reasons. The +DEFER modifier (originally created for deferred reads/writes - Section 4.2.7) is used as seen in Figure 4.7 and this time prevents the actual push taking place (although the stack pointer is still modified as if the push had been done).

```
DUP     SP( Int32 a - +DEFER Int32 a, Int32 a )
        IP( - next);
----------------------------------------
```

Figure 4.7: Stack push elimination in Tiger

4.2.6 Early Loading

An optimisation which can be turned on or off easily is *early loading*. This optimisation allows for the early loading of the address of the next opcode in the instruction stream

[3]Tiger actually provides an option eliminatePushes that can be set to on in order to emulate the behaviour of vmgen's stack push elimination scheme. However the +DEFER modifier is preferred because it is more explicit and can help avoid inadvertently eliminating critical stack pushes (for example where a popped item of the same name is being pushed back to the same slot, but has been modified by user-code in the meantime).

to a local variable, before the current opcode begins execution. All one needs to do is to turn the feature on if it is required. Tiger knows when to apply early loading (when the 'next' keyword is found in the instruction stream descriptor) to a generated VM instruction. It knows the location to read in (since Tiger tracks the instruction pointer) and finally it knows the type to read in (since it knows the base type for the instruction stream).

The next keyword is useful, because the instruction to be dispatched to is not always found in the instruction stream. For example, with relative branches, an offset will only be found in the instruction stream. A more extreme example would be an opcode that jumped to an address on the stack. In the absence of the next keyword, Tiger will not attempt to do any early loading. However, when the next keyword is present in the instruction stream behaviour, this indicates that the target of the dispatch at the end of the current instruction is to the next instruction in the instruction stream. In such circumstances (and when the earlyLoad option is on), Tiger generates code for early loading.

Figure 4.8 shows the effect of turning the early loading option on and off using a simple opcode definition. When early loading is off, it can be seen from the generated code that the address of the dispatch is only loaded immediately before the dispatch. When early loading is on, the address of the dispatch is loaded into a temporary variable (VM_EARLY_DEST), at the start of the opcode and the dispatch is to the address in this temporary variable at the end of the opcode.

4.2.7 Deferred Reading/Writing

Under certain circumstances, it may be desirable to prevent operands being read from the instruction stream unless they are actually required. A common example would be a conditional jump that only needs to read in the jump address/offset if the condition evaluates to the appropriate value. In Figure 4.9 we see an iflt opcode that jumps if the condition i1<0. The first version of iflt reads the offset to jump to from the instruction stream into the variable skip, regardless of whether the jump is to be taken or not.

In the second deferring version, skip is flagged with a +DEFER modifier in the instruction stream descriptor that indicates to Tiger that the user is taking the responsibility for loading skip from the instruction stream, if it is required. In order to assist

```
Tiger Code:     POP     SP( Int32 a -)
                        IP( - next )
                --------------------------------
```

```
Generated Code With Early Load Off:

                POP:
                {
                        Int32 a;
                        vm_Int32_equals_Int32(a,SPPTR[0]);
                        IPPTR=IPPTR+1;
                        SPPTR=SPPTR+1;
                        goto **IPPTR;
                }
```

```
Generated Code With Early Load On:

                POP:
                {
                        Int32 a;
                        void* VM_EARLY_DEST;
                        VM_EARLY_DEST=IPPTR[1];
                        vm_Int32_equals_Int32(a,SPPTR[0]);
                        IPPTR=IPPTR+1;
                        SPPTR=SPPTR+1;
                        goto *VM_EARLY_DEST;
                }
```

Figure 4.8: Early loading of dispatch address in Tiger

```
Without Defer:  iflt SP( Int32 i1 - )
                     IP( Int32 skip - next);

                if(i1 < 0)
                {
                        SET_IP(IPPTR+skip);
                }
                ----------------------------------------
```

```
With Defer:     iflt SP(  Int32 i1 - )
                     IP( +DEFER Int32 skip - +DEFER next);

                if(i1 < 0)
                {
                        VMLOAD_skip_;
                        SET_IP(IPPTR+skip);
                }
                VMLOADnext;
                ----------------------------------------
```

Figure 4.9: Deferred reading in Tiger

the user, Tiger declares the VMLOAD_offset_ macro automatically for any instructions that may require it. When executed, this macro will read in the value for offset. In this way we can prevent the unnecessary loading of operands, deferring them to a point in time where the user deems them necessary in the instruction.

Note the second deferred item, namely next, which is the address of the next instruction. Tiger allows this item to be deferred and when the address is required, the user can use a VMLOADnext macro which is defined automatically for them. A +DEFER modifier for next only has meaning when early loading (Section 4.2.6) is on. Otherwise the modifier is ignored and VMLOADnext is #defined to be nothing.

+DEFER works in the same way when applied to deferring popping items from the stack. In these cases, Tiger declares a VMREAD_*varname*_ macro instead of a VMLOAD_*varname*_ macro to prevent conflicts between stack reads and instruction stream reads. In addition, Tiger permits deferred writing to the stack, by using a +DEFER modifier in the push section of the stack behaviour. This time Tiger declares a VMWRITE_*varname*_ macro where required. Writing to the instruction stream (deferred or otherwise) is currently not permitted in Tiger.

4.2.8 Instruction Specialisation

The Java Language Specification [GJSB00], contains a number of specialised instructions. These are versions of instructions that already exist, but where the operands to the instruction have been 'hardwired' to a particular value. For example, the JVM contains an iload instruction which retrieves an integer value from a local variable and places it on the stack. The instruction takes one operand which identifies which local variable to use. The JVM also contains a specialised version of this same instruction, iload_0 which retrieves an integer value from local variable 0 and places it on the stack. This version of the iload instruction takes no operands since the local variable involved is known. For a more in-depth discussion of Instruction Specialisation, see Section 6.3.

Tiger allows the user to add specialised instructions to the instruction set through the use of a +SPEC keyword, allowing the user to define new instructions in the Tiger source file. Figure 4.10 shows the Tiger source for an unspecialised instruction ILOAD and the corresponding generated code. Compare this to Figure 4.11 where the definition for the specialised version is given. There are three points of note with respect to the

```
Tiger Code:     ILOAD:  SP( - Int32 a )
                        IP( Int32 index - next );
                        a=locals[index];
                ------------------------
```

```
Generated Code: ILOAD:
                {
                        Int32 a;
                        Int32 index;
                        vm_Int32_equals_Int32(index,IP[1]);
                        {
                                a=locals[index];
                        }
                        vm_Int32_equals_Int32(SP[-1],a);

                        SP+=(-1);
                        IP+=(2);
                        goto **IPPTR;
                }
```

Figure 4.10: Unspecialised ILOAD opcode

```
Tiger Code:     +SPEC ILOAD 7
```

```
Generated Code: #define index 7
                TIGER_SPECIAL_ILOAD_index_7:
                {
                        Int32 a;
                        {
                                a=locals[index];
                        }
                        vm_Int32_equals_Int32(SP[-1],a);

                        SP+=(-1);
                        IP+=(2);
                        goto **IPPTR;
                }
                #undef index
```

Figure 4.11: Specialised ILOAD opcode

Tiger source for the specialised instruction:

1. The actual definition for the ILOAD instruction has already been made. It does not need to be made again.

2. The name of the specialised instruction is not specified. It is generated automatically by Tiger.

3. The names of the operands are not given. Instead the operands to specialise are listed as per their order in the instruction stream. If some operands are not to be specialised a wildcard indicated by the '?' character can be used.

In the generated code for the specialised version, it can be seen that a macro replacement is used for specialising operands. Instead of declaring the variable index and reading its value from the instruction stream, a #define is used to define the token as a '7'. Apart from the #define, the omission of the declaration and reading of the value for index, the code remains the same as the unspecialised version of ILOAD. To avoid compiler warnings about redefinitions of various macros and to prevent accidental macro expansion elsewhere, there are #undefines at the end of any specialised instructions to remove any definitions for specialised operands which are no longer required.

In its present form, Tiger creates *non-compact specialisations*. In other words it will assume that even if an opcode has been specialised, the operand is still present in the instruction stream. Thus, both ILOAD and the specialised version of ILOAD take up the same amount of space in the instruction stream, although the latter ignores its operand. *Compact specialisations* present greater difficulties for the interpreter-writer due to the presence of code offsets (relative jumps/branches) in the bytecode. These code offsets must be fixed if redundant operands of specialised instructions are removed from the instruction stream.

4.2.9 Instruction Replication

Tiger provides support for the instruction replication optimisation (see Section 6.5). All one needs to do is to use the +ALIAS keyword in the Tiger source file. This keyword is then followed by the instruction which should be replicated and then by the number of times that it should be replicated. For example, Figure 4.12 shows how to create

```
Tiger Code:     +ALIAS POP 1;
```

```
Generated Code: POP_ALIAS_1:
                {
                        IPPTR=IPPTR+1;
                        SPPTR=SPPTR+1;
                        goto *opcodes[*IPPTR];
                }
```

Figure 4.12: Replicating the POP instruction

a single replica of the POP instruction. The replicas are named automatically, each one guaranteed a unique name. The actual code generated is identical to that of the original instruction which is being replicated, apart from the label which is unique for each replica.

4.2.10 Superinstructions

Superinstructions can be defined easily in Tiger by specifying the name of the desired superinstruction and the component instructions. The interpreter generator will then concatenate the component instructions in the correct order to create the superinstruction. In reality, the code that is generated is not just simply a concatenation of the component opcodes for several reasons:

1. In the superinstruction, updates to the instruction and stack pointers are combined, and only made when exiting the superinstruction.

2. Stack items are normally stored in local variables for the duration of the superinstruction. This *stack-slot caching* helps to speed up the superinstruction's access to stack items. Depending on the stack-slot caching mechanism used, changes to the stack are often not flushed until the superinstruction is being exited.

3. Tiger contains a mechanism for superinstructions across basic-blocks. In the case where there is a conditional branch, Tiger injects the code for updates to the stack and instruction pointers just before the branch. Also, if stack-slot caching is being used and some cached items need flushing, they will be flushed at this

point. Tiger can identify branches out of a superinstruction if they are flagged by a SET_IP macro.

Figure 4.13 shows how to define a new superinstruction super_1 in Tiger. Very little extra code is required in the source file, just the name for the new superinstruction and a list of the component instructions. Note that stack-slot caching is visible in the generated code.

Specialised Superinstructions

Tiger also permits the definition of specialised superinstructions. A specialised superinstruction is defined as a regular superinstruction, but any specialised operands are specified in the superinstruction definition. When a specialised superinstruction is being defined, it can be composed from a mixture of specialised instructions and non-specialised instructions.

Tiger does not require that specialisations for any of the specialised component instructions have been defined previously. If they are not present during specialised superinstruction creation, Tiger will create them automatically (Strictly speaking Tiger creates dummy versions termed *dummy specialisations* in cases like this. See Section 4.3.7 for details). Figure 4.14 shows an example of a simple specialised superinstruction being defined in Tiger. The instructions iload 6 and iload 7 are specialised component instructions, while aload is a non-specialised component instruction.

Stack-slot Caching Options

When concatenating a series of superinstructions, it can often be advantageous to load frequently used stack slots into local variables. Then any instructions which normally read from or write to the stack can read from or write to the local variables instead. At the end of the superinstruction (or any appropriate point in the superinstruction), any changes to the stack can be flushed as necessary by writing from the local variables to the stack. Tiger provides for a number of stack-slot caching mechanisms. The variations mostly relate to the scheduling of the caching and de-caching of stack slots throughout the superinstruction. They are:

1. **Off.** This is where no stack-slot caching is used. All reads and writes occur directly from/to the stack.

```
Tiger Code:     super_1 = dup ldc_quick;
```

```
Generated Code:

super_1:
{
    StackVal32 SPPTR_cached_minus1_;
    StackVal32 SPPTR_cached_0_;
    StackVal32 SPPTR_cached_1_;

    // dup
    SPPTR_cached_minus1_=SPPTR[-1];
    vm_StackVal32_equals_JavaVal32(SPPTR_cached_minus1_,s1)
    {
        JavaVal32 s1;
        vm_JavaVal32_equals_StackVal32(s1,SPPTR_cached_minus1_)
        vm_StackVal32_equals_JavaVal32(SPPTR_cached_0_,s1)
    }
    SPPTR[-1]=SPPTR_cached_minus1_;
    SPPTR[0]=SPPTR_cached_0_;


    // ldc_quick
    {
        JavaVal32 s1;
        vm_JavaVal32_equals_Address(s1,IPPTR[2])
        vm_StackVal32_equals_JavaVal32(SPPTR_cached_1_,s1)

    }
    SPPTR[1]=SPPTR_cached_1_;

    //Pointer updates and dispatch
    SPPTR=SPPTR+2;
    IPPTR=IPPTR+3;
    goto **IPPTR;
}
```

Figure 4.13: Defining a new superinstruction in Tiger

```
super_1 = iload 6 aload iload 7;
```

Figure 4.14: A specialised superinstruction in Tiger

2. **Simple.** This is where all stack-slots involved in the superinstruction are loaded into local variables at the start of the superinstruction. Any dirty cached stack-slots are written upon exiting the superinstruction.

3. **Conservative.** This approach is similar to the simple approach but delays caching stack-slots until they are needed and de-caches them as soon as they are no longer needed. This is a useful approach to reduce register pressure. This approach adds an additional constraint in that the reading and writing of stack items must not violate the stack paradigm. For example the reading (caching) of a slot under the topmost slot of the cache, followed by the reading of the topmost slot of the stack is not permitted. The other way around, however would be fine.

4. **Aggressive.** This approach is identical to the conservative option, but dispenses with the any ordering rules relating to the stack paradigm. Reads (caches) take place as late as possible and writes (de-caches where the item is dirty) take place as early as possible)

The choice of stack-slot caching mechanism is made by setting the option stackUpdate-Combining to the desired value (off, simple, conservative or aggressive).

4.2.11 Preferred Instructions

Some instructions are unsuitable as components for superinstructions. For example when stack-slot caching is on (i.e. when stack slots are stored in local variables), an instruction that performs an optimised direct memory to memory copy, for example to duplicate an item on the stack would cause difficulties. This could be solved by rewriting the instruction to get it to read the item to be duplicated from the stack and then to write it back in the normal way. This latter approach has the benefit that it could be incorporated into a superinstruction, but as a standalone instruction, would not be as optimised as the direct memory-to-memory version which cannot be included in superinstructions.

In order to get the best of both worlds, Tiger permits what are termed *preferred instructions* which are versions of instructions that are optimised for standalone usage (ie outside superinstructions). *Non-preferred instructions* are versions of the same instruction which are intended for usage only inside superinstructions. In this way, two versions of the same instruction can be specified by the user, one for inclusion in superinstructions, and one for standalone execution.

```
ldc2_w_quick  SP( - JavaVal32 s1,JavaVal32 s2)
              IP( JavaVal32 s1, JavaVal32 s2 - next);
-----------------------------------------------
+PREF ldc2_w_quick
              SP(- +DEFER JavaVal32 s1,+DEFER JavaVal32 s2)
              IP( +DEFER JavaVal32 s1, +DEFER JavaVal32 s2 - next);
              memcpy64(SPPTR,(Uint32*)(IPPTR+1));
-----------------------------------------------
```

Figure 4.15: Preferred and non-preferred instructions in Tiger

In Figure 4.15, an example can be seen where preferred instructions might be useful. The non-preferred version of ldc2_w_quick leaves it up to Tiger to generate the appropriate code to read from the instruction stream and push to the stack. On the other hand, the preferred version of ldc2_w_quick uses the +DEFER keyword to indicate to Tiger that the reads and writes are to be handled by the user (although Tiger will still generate the correct stack and instruction pointer updates at the end of the instruction). Inside the preferred version of ldc2_w_quick a 64-bit copy macro is used to copy directly from the instruction stream onto the stack, which may be efficient but which is unsafe inside a superinstruction.

4.3 Tiger - Back-end Functionality and Requirements

In the previous section, we saw an array of optimisations and code generation techniques from the point of view of the Tiger source code file. Although it has been shown that Tiger permits the addition of new instructions such as superinstructions, replica-

tions and specialisations, we have not yet discussed what mechanisms Tiger provides for simplifying the incorporation of these new instruction into an existing interpreter. In this section we focus on what the user has to do in order to integrate Tiger with their interpreter and the features that Tiger provides to aid in this integration.

4.3.1 Generated Interpreter Core

A source input file to Tiger is of the form *filename*.vmj. When Tiger is run on this file, it generates a 'C' file, *filename*-vm.i which contains the code for the interpreter core. The user must include this file in the appropriate section of their interpreter (simply by using a #include). All the VM instruction implementations (including superinstructions, replications and specialisations) are present in this interpreter core file. In a switch-based interpreter (Section 3.1) each instruction is preceded by a case statement and ended by a break whereas in the case of a token threaded (Section 3.2.2) or a direct threaded (Section 3.2.1) interpreter, each instruction begins with a label and ends with the appropriate dispatch.

4.3.2 User-Supplied Type Conversion Macros

In the Tiger source file, the types for the stack and instruction stream are specified. Any types that are stored in the instruction stream or stack are also listed in the Tiger source file (Section 4.2.1) with the intention of telling Tiger how many slots each of these types take up in the stream or stack. For example, in an interpreter with a stack type of char, if a 32-bit value was to be stored on the stack, then that 32-bit type must be registered with a size of 4.

Although no explicit stack or stream related type conversions are normally apparent in the Tiger source file, many are automatically created in the generated code. For example, every time type A is read from a stack or stream of type B, Tiger generates a vm_A_equals_B(*dest*,*source*) macro. Similarly when type A is to be written to a stack (no writing to instruction streams is currently permitted) of type B, a vm_B_equals_A(*dest*,*source*) macro is used by Tiger. These macros need to be supplied by the user and included in the interpreter. Each of these macros performs an assignment from *source* to *dest* (with the appropriate type conversions).

There are two important notes with respect to these type conversions. Firstly, even when the *source* and *dest* are the same type, Tiger will use a macro to perform the assignment. For example, if the type of the stack is A and we are reading an item of type A from the stack, a vm_A_equals_A(*dest*,*source*) macro will be used. This can be useful in situations where A is a complex data type and might, for example, require a deep copy for the assignment to be carried out correctly. Secondly, because multiple stack or instruction slots may be involved when reading or writing larger data types, these macros can often take multiple arguments for the source or destination, one for each slot. For example, when reading a 32-bit integer from an 8-bit stack, the macro used would be vm_Int32_equals_char(*dest*,$source_0$,$source_1$,$source_2$,$source_3$) where $source_0$ to $source_3$ are the stack slots holding the four bytes of the 32-bit integer. Figure 4.16 shows the generated code for the JVM instruction dload. The instruction stores a 64-bit type JavaDouble into a stack composed of slots of a 32-bit type, StackVal32. Therefore two stack slots are required for the destination in the vm_StackVal32_equals_JavaDouble macro.

```
dload:
{
    JavaDouble jd1;
    Uint8 i;
    vm_Uint8_equals_Address(i,IPPTR[1]);
    {
        jd1=jvm2Double(&locals[i]);
        vm_StackVal32_equals_JavaDouble(SPPTR[0],SPPTR[1],jd1);
    }
    IPPTR=IPPTR+2;
    SPPTR=SPPTR+2;
    goto **IPPTR;
}
```

Figure 4.16: Multiple slot type conversions in Tiger

4.3.3 Instruction Indices

Each instruction generated by Tiger, whether it be a standard instruction, a specialised instruction, a superinstruction, a replicated instruction or any combination of these,

is assigned a unique integer, its index, by Tiger. This index is critical to many functions of the interpreter including parsing for superinstructions and also for obtaining the address of an instruction for token threaded dispatch (in combination with the instruction labels file - Section 3.2.2).

The indices are numbered contiguously from 0 and are to be found in the automatically generated file *filename*-indices.i. The grouping of indices can be seen in Figure 4.17. This grouping permits the tables for parsing superinstructions to be as compact

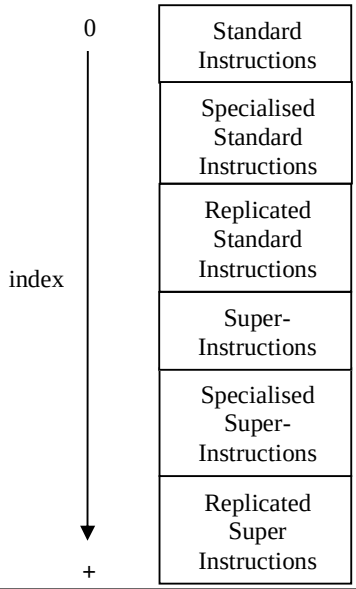


Figure 4.17: Indices in Tiger

as possible. An example of the indices file can be seen in Figure 4.18.

```
#define INDEX_OF(a) tiger_prim_index_##a
#define tiger_prim_index_aastore 0
#define tiger_prim_index_aconst_null 1
#define tiger_prim_index_aldc_ind_quick 2
#define tiger_prim_index_ldc_quick 3
#define tiger_prim_index_ldc2_w_quick 4
......
```

Figure 4.18: The Tiger-generated index file

Note the INDEX_OF macro which uses token pasting to allow us to refer to the indices of instructions in a simplified manner e.g. INDEX_OF(aastore).

4.3.4 The Labels Array

In addition to generating a *filename*-indices.i file, Tiger also generates another file *filename*-labels.i which contains an array of labels for all of the instructions Tiger has created. There is a direct one-to-one correspondence between the index of an instruction and its position in the labels array. Specifically, the index of the label of an instruction in the labels array is the same as the Tiger-assigned index for that instruction, defined in the file *filename*-indices.i. Sample contents for the *filename*-labels.i file and how to include the array in the interpreter can be seen in Figure 4.19. Two aspects

```
labels file:    INST_ADDR(aastore),
                INST_ADDR(aconst_null),
                INST_ADDR(aldc_ind_quick),
                INST_ADDR(ldc_quick),
                INST_ADDR(ldc2_w_quick),
                ......
```

```
usage:          void* mylabels=
                {
                    #include "cvm-labels.i"
                };
```

Figure 4.19: Using Tiger-generated labels

of this labels file are noteworthy:

1. The file only contains the contents of the array. If the user wants to create an array using the labels file, they must choose an appropriate name for their array and include the labels file in the correct place.

2. The labels file uses an INSTR_ADDR macro to refer to the address of an instruction. This macro, defined by Tiger, uses token-pasting to expand the argument out to an expression representing the address of the label for that instruction. For GCC's labels-as-values syntax, INSTR_ADDR(instr) expands to &&instr.

The labels file is extremely useful during the translation process when implementing a direct threaded interpreter or when performing a dispatch when using token-threaded dispatch.

It should be noted that being able to store labels (or addresses of labels) is a GCC extension termed *labels-as-values*. The ANSI-C standard does not support this extension, although it can be found in other compilers such as Intel's icc compiler [Int04].

4.3.5 Instruction Names

To assist in debugging and profiling, Tiger also generates a typed, named and initialised array in a file called *filename*-names.i. The name for this array is defined in Tiger as VM_OPCODE_NAMES, with a type of char*. An example of the contents of this file can be seen in Figure 4.20.

```
#define VM_OPCODE_NAMES_COUNT 610
char* VM_OPCODE_NAMES[VM_OPCODE_NAMES_COUNT]={
"aastore",
"aconst_null",
"aldc_ind_quick",
"ldc_quick",
"ldc2_w_quick",
......
```

Figure 4.20: The Tiger-generated names file

4.3.6 Replication File

A useful capability of Tiger is the automatic creation of a file called *filename*-alias.i which consists of a set of macros to enable the support of replicated instructions. This file contains the definition for ALIASED_INDEX_OF(instr_index) which enables an instruction with an index of instr_index to be replicated. This macro provides support for round-robin replication (Section 6.5) so that replications, if they exist, are issued in a cyclical order. In Figure 4.21, you can see the definition for the ALIASED_INDEX_OF macro. In this example, there are 6 copies of aastore, 5 copies

```
#define ALIASED_INDEX_OF(opcname) aliased_vm_index_##opcname

static int vm_ptr_alias[]={0,0,0};
const static int vm_alias_bipush[]={5,210,211,212};
const static int vm_alias_aastore[]={0,213,214,215,216,217};
const static int vm_alias_aconst_null[]={1,218,219,220,221};

#define aliased_vm_index_aastore vm_alias_aastore[vm_ptr_alias[1]=(vm_ptr_alias[1]+1)%6]
#define aliased_vm_index_aconst_null vm_alias_aconst_null[vm_ptr_alias[2]=(vm_ptr_alias[2]+1)%5]
#define aliased_vm_index_aldc_ind_quick 2
#define aliased_vm_index_ldc_quick 3
#define aliased_vm_index_ldc2_w_quick 4
#define aliased_vm_index_bipush vm_alias_bipush[vm_ptr_alias[0]=(vm_ptr_alias[0]+1)%4]
......
```

Figure 4.21: A Tiger-generated replication file

of aconst_null and 4 copies of bipush. For each replicatable instruction *instr*, an array vm_alias_*instr*[] of the indices of all replications of that instruction is declared in the replication file. For example, vm_alias_bipush[] is declared and initialised to be a list of the indices of all copies of bipush. Additionally, there is an array of indices, vm_ptr_alias, containing one element for each replicatable instruction. In the given example, vm_ptr_alias[0] is used to point to the element of vm_alias_bipush that will be returned the next time ALIASED_INDEX_OF(bipush) is called. The effect of calling ALIASED_INDEX_OF(bipush) will be to add to vm_ptr_alias[0] (or reset it to 0) and to return the index of the next copy of bipush.

For a non-replicated instruction *instr*, calling ALIASED_INDEX_OF(*instr*) equates to aliased_vm_index_*instr* which in turn equates to the index of *instr*. So effectively, calling ALIASED_INDEX_OF(*instr*) for a non-replicated instruction is equivalent to INDEX_OF(*instr*).

4.3.7 Specialisation File

In order to support any generated specialised instructions, Tiger generates a file *filename*-special.i. This file contains a number of automatically-generated macros which enable specialisation. For each instruction *instr* an automatically-generated macro vm_specialise_*instr*(...) is defined in the file. This specialisation macro takes a variable number of arguments representing specialisable operands in the instruction stream. After determining if instr can be specialised (by inspecting the operands supplied), the

macro then evaluates to the index of the specialised instruction if one was found or to the index of instr if no specialisation was found.

Strictly speaking, the macro tries to alias the instruction through the replication macros (in *filename*-alias.i) after it attempts to find a replication, and so the macro can evaluate to an aliased instruction.

```
#define vm_specialise(_opcode,...) vm_specialise_##_opcode(__VA_ARGS__)
#define vm_specialise_aastore(...) \
                (INDEX_OF(aastore),ALIASED_INDEX_OF(aastore))
#define vm_specialise_aconst_null(...) \
                (INDEX_OF(aconst_null),ALIASED_INDEX_OF(aconst_null))
#define vm_specialise_bipush(_value0)\
                ((((_value0)==4))? (INDEX_OF(vm_spec_bipush_i_4),\
                                    ALIASED_INDEX_OF(vm_spec_bipush_i_4))\
                    :  ((((_value0)==7))? (INDEX_OF(vm_spec_bipush_i_7),\
                                    ALIASED_INDEX_OF(vm_spec_bipush_i_7))\
                        : (INDEX_OF(bipush),ALIASED_INDEX_OF(bipush))))
......
```

Figure 4.22: A Tiger specialisation file

Figure 4.22 shows a sample of a typical *filename*-special.i file. In the example, the aastore and aconst_null instructions have no specialisation associated with them. A call to vm_specialise_aastore or vm_specialise_aconst_null will simply attempt to alias the instruction. On the other hand, there are two specialisations for bipush (bipush 4 and also bipush 7). The macro vm_specialise_bipush examines the supplied operand using a cascaded conditional statement and returns an alias of either vm_spec_bipush_i_4, vm_spec_bipush_i_7 or simply bipush in the case where there is no specialisation for that operand with bipush.

The cascaded conditional approach works quite well in practice but for large numbers of specialisations it may be unsuitable, due to the possibly high number of comparisons required to determine if an instruction is specialisable. In such circumstances, a hash-table approach might be a more efficient approach. More advanced compilers may actually perform the transformation from cascaded conditional to hash-table automatically. Alternatively, the cascaded conditional could be rephrased as a switch-statement, which in turn might be better optimised by the C compiler.

Dummy Specialisations

As noted in Section 4.2.10, when an attempt to specialise a superinstruction takes place, Tiger checks to see if the individual specialised instructions that make up the specialised superinstruction actually exist. If they do, Tiger has nothing extra to do. If not, Tiger has to create them.

In order to avoid creating numerous specialised instructions purely to support the creation of specialised superinstructions, Tiger creates *dummy specialisations*. These dummy specialisations are different to regular specialised instructions because there is no unique corresponding instruction emitted by Tiger for the dummy specialisation. However, a dummy specialisation does get assigned an instruction index which is used later during the superinstruction parsing process. When a dummy specialisation is encountered in the instruction stream, the instruction that is executed is the original non-specialised instruction. For example if vm_spec_bipush_i_9 was a dummy specialisation, then when it is encountered in the instruction stream, the actual instruction that will get executed will be bipush since there is no implementation for the dummy specialisation.

4.3.8 Superinstruction Parsing

In order to support parsing bytecode for superinstructions (Section 6.4.3), Tiger creates a *filename*-parse.i file. This file contains hash-tables for a unified Deterministic Finite-State Automata (DFA) for all superinstructions, with accepting states corresponding to a successful parse to a superinstruction. In order to reduce the number of transitions in the DFA, nodes with single transitions between them are unified. An important aspect of the DFA that Tiger generates, is that it is designed for parsing backwards through a method.

Figure 4.23 illustrates a simple DFA with unified single-transition states. The numbers represent the indices for component instructions of the DFA. The states with a thicker solid line around them are final states that emit a superinstruction index, when entered. Note how two single transitions have been absorbed into state **5**, thereby eliminating two other states. Once state **5** is entered, it will only be exited when the symbols *7* and *6* and then either *5* (leading to state **-1**) or *7* (leading to state **-2**) are read in.

```
indices:        #define INDEX_OF(a) tiger_prim_index_##a
                ......
                #define tiger_prim_index_bipush 5
                #define tiger_prim_index_sipush 6
                #define tiger_prim_index_fload  7
                ......
                #define tiger_prim_index_super_0 210
                #define tiger_prim_index_super_1 211
                #define tiger_prim_index_super_2 212
                #define tiger_prim_index_super_3 213
                ......
```

```
superinstrs:    super_0 = bipush sipush fload bipush;
                super_1 = fload sipush fload bipush;
                super_2 = fload sipush;
                super_3 = bipush sipush;
```

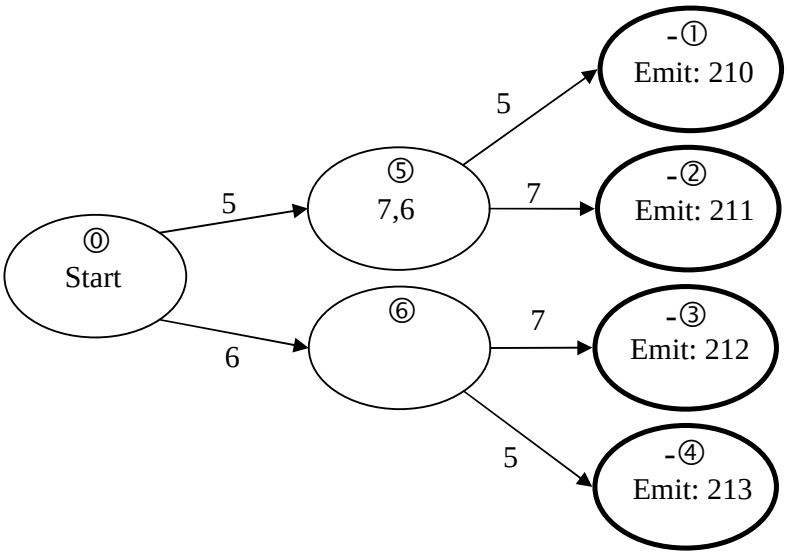


Figure 4.23: A sample superinstruction-parsing DFA in Tiger

Although the idea of using a collapsed-state DFA is relatively straightforward, the data structures in the *filename*-parse.i file require some explanation. In order to be as efficient as possible, the parse file consists of a number of hash-tables so that, during a superinstruction parse, one simply needs to hash on the current state and symbol (instruction) to get the next state. Each state gets its own hash-table so in reality, when one has a symbol/state pair, it is necessary to first find the appropriate hash table using the state and then hash using the symbol to get the new state.

Collapsed states are handled differently, since they contain internal transitions. Each collapsed state has an internal pointer to keep track of what the next required internal symbol is. If the new symbol read by the DFA does not match the required symbol, then that DFA stops. If the symbol matches the next required symbol, this internal pointer is moved on to the next symbol. If there is no next internal symbol, the next symbol to be read will result in a transition to a new state or the termination of that DFA.

```
#define VM_SUPER_START 210
#define VM_MAX_LEN 4
#define VM_SUPER_COUNT 4
static int mergedSizes[]={0,0,0,0,0,2,0};
static int mergedOffsets[]={0,0,0,0,0,2,2};
static int mergedSymbols[]={7,6};
static const Int16 entryPoints[]={-1,0,0,0,0,-5,-4,0};
static const node sharedTable[]={{5,-1},{6,-4},{5,-2},{6,-3},{0,5},{0,6}};
#define VM_SHAREDTABLESIZE 6
static int vm_codes[]={0,210,211,212,213};
#define VM_MAXSTATES 7
```

Figure 4.24: A sample DFA-based parsing file in Tiger

In Figure 4.24 the corresponding Tiger-generated parse file for the superinstructions in Figure 4.23 can be seen. Some of the simpler items to describe in this file are VM_SUPER_START (the index of the first superinstruction), VM_MAX_LEN (the length of the longest superinstruction) and VM_SUPER_COUNT (the total number of superinstructions).

Next, we consider how collapsed-states are implemented using the mergedSizes,

mergedOffsets and mergedSymbols arrays. For each state i mergedSizes[i] holds the number of internal transitions for that state (0 for non-collapsed states with no internal transitions). The value stored in mergedOffsets[i] gives the index in the mergedSymbols[] array where the list of symbols for internal transitions occur. You can see in the example that mergedSizes[5] is 2. The entry in mergedOffsets[5] is 0. Therefore if we look at mergedSymbols[0] we will find the first symbol corresponding to the first internal transition for state 5. This is followed by the next symbol mergedSymbols[1] required for the next internal transition for state 5.

Construction of the Parse File

Much of the contents of the parse file are straightforward to implement. However, the motivations and mechanism for the shared hash-table deserve some explanation. The most important feature required from the hash, as with any hash, is speed. Considering a hash-table for a single state i, that hash-table must store the new state entered when a symbol s is read. Given that the indices of states in Tiger are contiguously allocated and represent a reasonably tight range of numbers, there is no real need for a modulus operator to be applied to the symbol (i.e. the index of an instruction). The hash lookup is simply to use the symbol s as an index into an array representing the hash-table. If that entry j in the array is non-zero, then there is a transition {i,s}→j. This type of hash is perfect but quite sparse (i.e. most entries in the array will be 0).

In order to compact the sparse hash tables, they are all consolidated or rather overlaid onto a single hash table. Because the overlaying procedure can move a hash table, a separate array (entryPoints) is required to keep track where the hash table for each state begins in the shared hash table. Each entry in the shared hash table is now an ordered pair consisting of the owning state and the original contents of the owning state's hash table in that order. Figure 4.25 shows a small example, where three hash tables are overlaid on top of each other. Note how there are no gaps in the shared hash table. This is not always the case, but Tiger does attempt to minimise the sparseness of the shared hash table. (See Appendix 8.6 for more details).

The data structures supplied by Tiger to assist with the parsing of bytecode do not force the interpreter writer to implement the parse in either a greedy or optimal fashion (see Section 6.4.3).

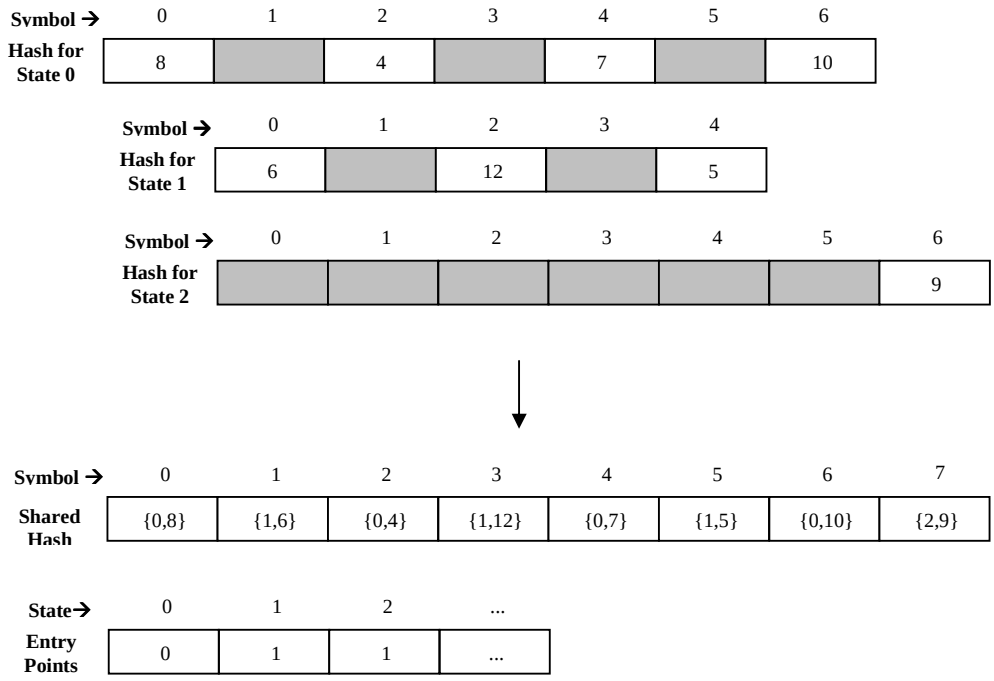


Figure 4.25: Overlaying of Hash Tables in Tiger

4.3.9 Global Definitions

In addition to the files above, Tiger also generates a *filename*-globals.i file which contains a number of miscellaneous definitions. For each of the *option-value* statements in the interpreter source file (Section 4.2.1), a symbol VM_OPTION_*option_value* is defined in the global definitions file. This can be useful where the interpreter writer wants to have additional code in their interpreter only when a certain option is set in Tiger. In essence it provides a method for the interpreter writer to determine which options have been set in Tiger.

In addition, Tiger also declares a number of interpreter and thread start-up and shut-down macros which should be invoked at the appropriate time by the interpreter-writer. These macros are useful in a number of circumstances such as various profiling operations which need to be initialised on startup and have their data written on termination.

4.4 Interpreter Diagnostics

When generating code, Tiger can insert certain kinds of diagnostic code, depending on which options are set in the Tiger source file. In this section, we examine each of them in turn, including how to set them, how Tiger implements them, and how to understand the data that is generated.

4.4.1 Histogram

The most simple of the diagnostics is the histogram. This histogram option inserts code into the generated interpreter core that increments a counter associated with each instruction, each time that instruction is executed. When the interpreter terminates the histogram data is written to a file containing these counts. To turn on the histogram generation the *histogram* option is set to *on* in the Tiger source file. This triggers some changes in the way Tiger generates code. Specifically, the VM startup macro is modified to declare and initialise the histogram, while the VM shutdown macro is modified to write the histogram data to the file *histogram.dat*. In addition, a small amount of code is added into the implementation of each instruction implementation in the interpreter core, to modify the histogram counts each time the instruction is executed.

Figure 4.26 illustrates sample contents of this histogram file. The contents are sorted according to instruction index (in increasing order). A Java tool, *BarChart*, is supplied with Tiger that reads in this file and generates a Scalable Vector Graphic [TB02] representation of the histogram. Figure 4.27 shows sample output for the tool.

```
......
DUP     30915493
DEREF   98972729
POSTINC 7015269
POSTDEC 14952
POP     41926355
ASSIGN  15501107
EXIT    1
PUSHAL  102705447
PUSHAA  2406
PUSHAG  18794450
PUSHAC  60960609
ALLOC   0
JUMP    7608795
JFALSE  64735992
PUSHS   6
RETURN  12
CALL    1877
......
```

Figure 4.26: Sample from *histogram.dat*

4.4.2 Dispatch Tracking

Much of the research for which Tiger was developed has focussed on the effect and elimination of branch mispredictions in interpreters. In order to instrument interpreters more comprehensively, Tiger offers an option branchData that, when set to on, tracks the calling sequence for each dispatch in the interpreter core.

In its present form, branchData tracking adds code to each dispatch point in the generated interpreter. This extra code records the target each time the dispatch is invoked and increments a misprediction estimation counter every time the current target differs from the target the last time this dispatch was invoked. Tiger can track

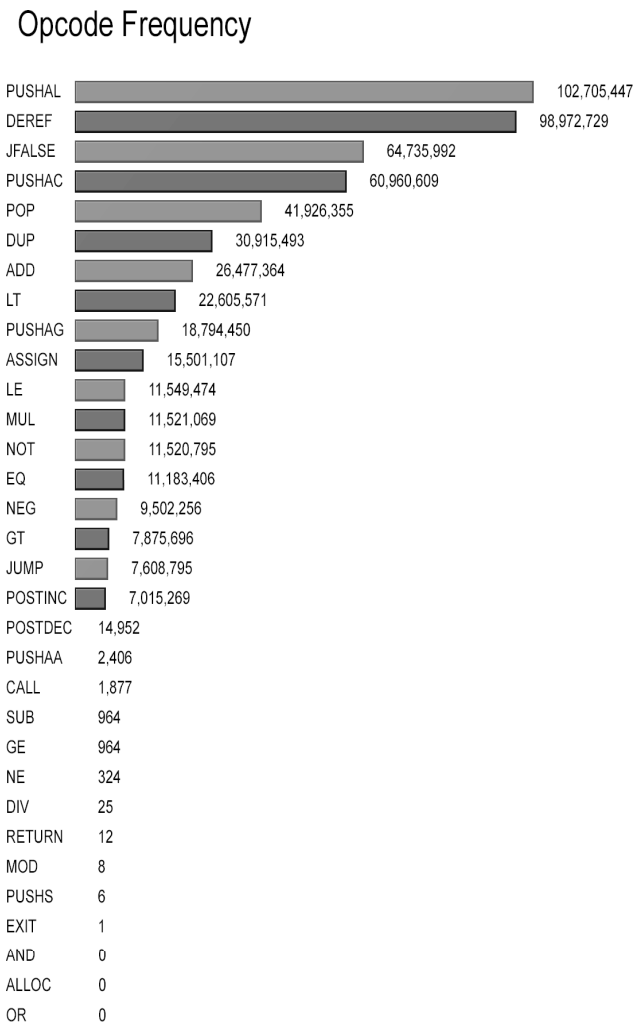


Figure 4.27: SVG representation of *histogram.dat*

multiple dispatch points inside a single instruction, as might occur in an iflt instruction (Figure 4.9).

In addition to estimating the mispredictions on a per-branch basis, Tiger also records a histogram of all targets for each branch. Thus for each branch we get:

1. An estimation of the number of mispredictions at that point.

2. A histogram of all targets jumped to from that point.

The misprediction rate, although at first glance naive, is a good estimator (at least on a global level), even for architectures with more sophisticated branch prediction algorithms. Preliminary experiments with the Pentium 4 using the misprediction rate from the hardware performance counters in comparison to the global estimated misprediction count from Tiger (i.e. the sum of all misprediction counts for all dispatch points) suggest that the Tiger estimate is within 10% of the real value.

The real value of the branchData option is that it does give a more fine-grained measure of mispredictability than the hardware performance counters. This information could then be fed back into more advanced static replication schemes that select replications based on mispredictability rather than static occurrences.

Figure 4.28 shows the output of the branchData option, which can be found written to the *branchdata.dat* file after the interpreter has terminated. For each dispatch point, various information is recorded in the output file:

1. **Source**. The address in the memory where the dispatch lies.

2. **Owner**. The name of the instruction to which the dispatch belongs.

3. **Ref**. The address in the memory where the dispatch lies.

4. **Mispred**. The estimate for branch mispredictions at this dispatch.

5. **Count**. The total number of times this dispatch point has been used.

6. **Rate**. This is the misprediction estimate divided by the count.

7. **Dispatch Target Information**. For each new dispatch target from the current dispatch point, the following is recorded:

```
......
source:0x804b9ec owner:JUMP ref:745 mispred:59688 count:7604668 rate:0.007849
    target:0x804b9b2 name:JUMP count:29840
    target:0x804bbd6 name:RETURN count:1
    target:0x804b745 name:PUSHAA count:103
    target:0x804b6a5 name:PUSHAL count:7574724
source:0x804ba94 owner:JFALSE ref:779 mispred:25012008 count:27335561 rate:0.914999
    target:0x804b7d4 name:PUSHAG count:3555803
    target:0x804b9b2 name:JUMP count:2716578
    target:0x804bbd6 name:RETURN count:9
    target:0x804b745 name:PUSHAA count:276
    target:0x804b3a6 name:DUP count:5694211
    target:0x804ba32 name:JFALSE count:5825348
    target:0x804b863 name:PUSHAC count:11
    target:0x804b6a5 name:PUSHAL count:9543325
source:0x804bae8 owner:JFALSE ref:790 mispred:22900937 count:37389283 rate:0.612500
    target:0x804b7d4 name:PUSHAG count:3719088
    target:0x804b745 name:PUSHAA count:867
    target:0x804b581 name:POP count:19394695
    target:0x804b863 name:PUSHAC count:66
    target:0x804b6a5 name:PUSHAL count:14274567
source:0x804bb90 owner:PUSHS ref:820 mispred:0 count:6 rate:0.000000
    target:0x804bcb0 name:CALL count:6
......
```

Figure 4.28: Sample from *branchdata.dat*

7a. Target The address to which the dispatch is going.

7b. Name The name of the instruction at that point.

7c. Count The number of times to which the dispatch has jumped to that point.

Tiger is supplied with a Java tool, DispatchGraph, which creates a Scalable Vector Graphic (SVG) dispatch-graph from the generated file *branchdata.dat* by using the GraphViz [EGK+02] package. There are various options for graph generation, the most important of which is the specification of a minimum threshold of dispatches which must be reached before a dispatch edge is drawn between two nodes. This is quite useful when one wishes to eliminate less frequent dispatches in order to simplify the visual representation. Nodes with no edges after the application of the threshold are eliminated. An example of the output of this tool can be seen in Figure 4.29.

Note the way in which the two dispatch points for JFALSE are aggregated into a single node representing the entire JFALSE instruction. Each node in the graph represents the implementation of an instruction in the interpreter core. Each box in a node represents a dispatch point in that node. The first number indicates the number of times that dispatch has been invoked. Alongside this number is the percentage of

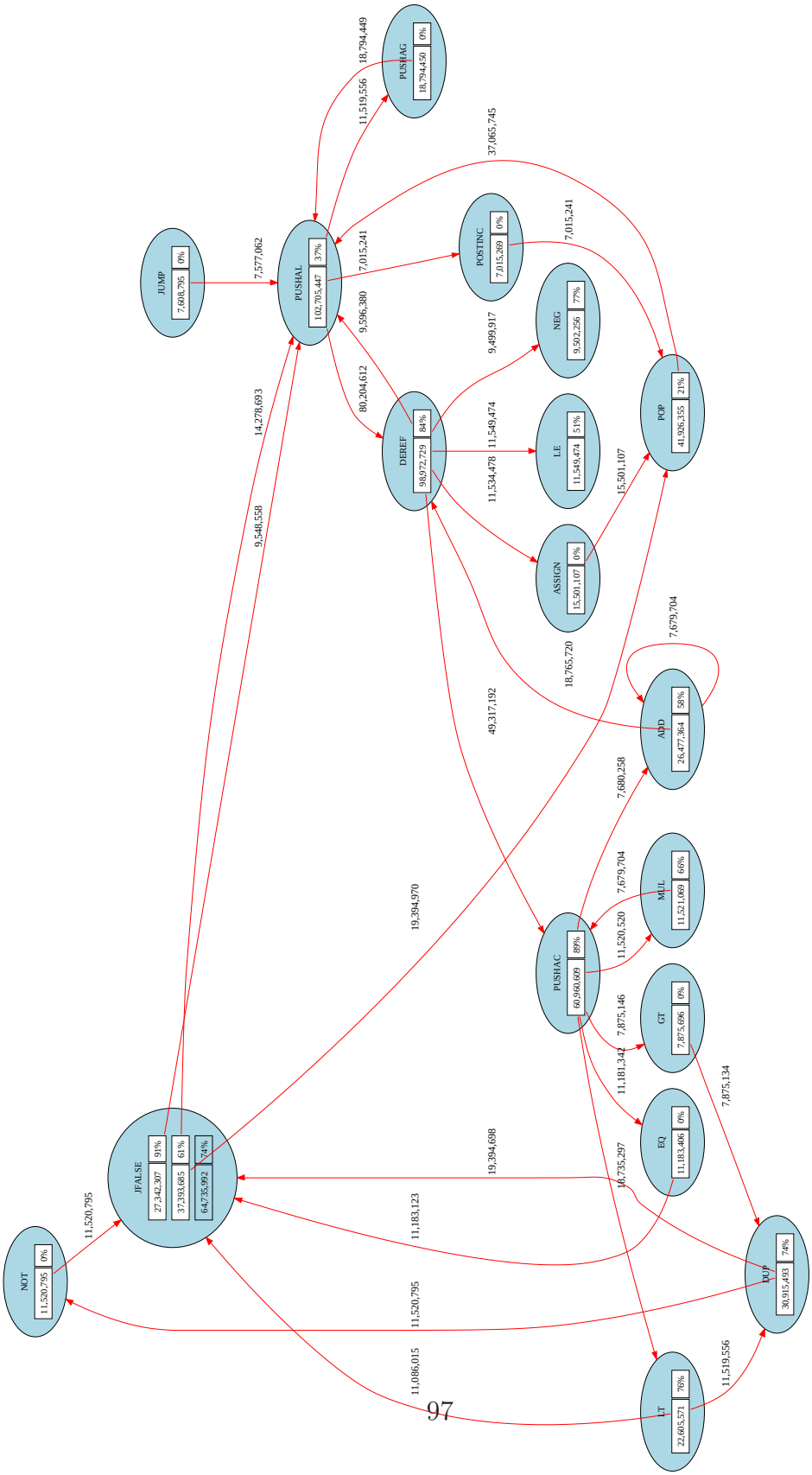


Figure 4.29: SVG representation of *branchdata.dat*

mispredictions for that dispatch point. Nodes with multiple dispatch points also have a total sum of the number of times all dispatches in that node have been invoked, along with the average misprediction rate across all those dispatch points. Directed edges from dispatch points to instructions represent a dispatch from the source point to the target point (an instruction). These edges are labelled with the number of times that particular dispatch jumped to the target instruction. Adding up the weights of all edges out of a dispatch point does not always add up to the total number of dispatches at that point. This is due to the minimum threshold of times that a dispatch must be made from a dispatch point before the edge gets displayed (a user-specified argument to the DispatchGraph tool).

The support code in the global definitions file generated by Tiger to deal with branchData is shown in Figure 4.30.

```
......
#include "support/branchData.h"

#define VM_BRANCHLABEL         VM_BRANCHLABEL2(__LINE__)
#define VM_BRANCHLABEL2(a)     VM_BRANCHLABEL3(a)
#define VM_BRANCHLABEL3(a)     VM_BRANCHLABEL_##a:
#define VM_ADDRESSOFBRANCH     VM_ADDRESSOFBRANCH2(__LINE__)
#define VM_ADDRESSOFBRANCH2(a) VM_ADDRESSOFBRANCH3(a)
#define VM_ADDRESSOFBRANCH3(a) &&VM_BRANCHLABEL_##a
#define VM_BRANCH(a) VM_BRANCHLABEL \
               vmjen_branch_process(VM_CURRENT_OPCODE,__LINE__,(a),VM_ADDRESSOFBRANCH);

//-----------------------------------------------------

#define VM_ON_EXIT  vm_branch_dump(VM_OPCODE_NAMES,VM_OPCODE_COUNT,VM_LABELS);
#define VM_ON_ENTRY vm_branch_init();

#define VM_THREAD_ON_EXIT
#define VM_THREAD_ON_ENTRY
......
```

Figure 4.30: Global definitions to support the branchData option

The main features are the inclusion of the *branchData.h* file, the initialisation of the appropriate data-structures in VM_ON_ENTRY (vmjen_branch_init) and the writing of all the collected branch data in VM_ON_EXIT (vmjen_branch_dump). Also the VM_BRANCH macro (and supporting macros) is defined here. The inclusion of this macro in the generated interpreter core and how it is expanded by the 'C' preprocessor can be seen in Figure 4.31.

Note in Figure 4.31 how the macro expands to a call to vm_branch_process, which is

Before Expansion:

```
......
#define VM_CURRENT_OPCODE "JFALSE"
JFALSE:
{
    Int32 a;
    short b;
    vm_short_equals_char(b,IP[2],IP[1]);
    vm_Int32_equals_Int32(a,SP[0]);
    {
        if(a)
        { ; }
        else
        {
            IP=((mem+b));
            SP+=(1);
            VM_BRANCH(VM_LABELS[IP[0]]);
            goto *VM_LABELS[IP[0]] ;
        }
    }
    SP+=(1);
    IP+=(3);

    VM_BRANCH(VM_LABELS[IP[0]]);
    goto *VM_LABELS[IP[0]];
}
#undef VM_CURRENT_OPCODE
......
```

After Expansion:

```
......
JFALSE:
{
    Int32 a;
    short b;
    vm_short_equals_char(b,IP[2],IP[1]);
    vm_Int32_equals_Int32(a,SP[0]);
    {
        if(a)
        { ; }
        else
        {
            IP=((mem+b));
            SP+=(1);
            VM_BRANCHLABEL_788:
                vm_branch_process("JFALSE",788,(VM_LABELS[IP[0]]),&&VM_BRANCHLABEL_788);
            goto *VM_LABELS[IP[0]] ;
        }
    }
    SP+=(1);
    IP+=(3);

    VM_BRANCHLABEL_796:
        vm_branch_process("JFALSE",796,(VM_LABELS[IP[0]]),&&VM_BRANCHLABEL_796);
    goto *VM_LABELS[IP[0]];
}
```

Figure 4.31: Sample from generated interpreter core with branchData option

the function that records all information associated with the branch. The most critical aspect of this macro is the automatic creation of a label VM_BRANCHLABEL_*lineno* where *lineno* is the line on which the branch occurs. This facilitates the unique identification of each branch that is being profiled.

4.4.3 Debugger

Tiger provides a debugger option that, when set to on, inserts code into the interpreter core. For each instruction, this additional code dumps the instruction name to *stderr*, along with any operands read from the instruction stream. Any items that are read from or written to the stack are also written to *stderr*. The output of the debugger can be seen in Figure 4.32. A section of the additional code generated by Tiger can be

```
......
ifeq
 SP:pop:i1:=0
 IP:read:skip:=8
monitorexit
 SP:pop:directObj:=0x824e2dc
goto
invokenonvirtual_quick
 IP:read:tmb:=0x8269184
 IP:read:tcb:=0x8156d80
ifnonnull
 SP:pop:o1:=(nil)
fload
 IP:read:i1:=5
 SP:push:f1:=(nil)
return
......
```

Figure 4.32: Sample debugger output

found in Figure 4.33.

For each type read from the instruction stream or stack and each type written to the stack, the interpreter-writer must supply a vm_dump_type(*inst*) macro that writes an instance *inst* of that type to *stderr*. Figure 4.33 illustrates some sample definitions. These macros must be included in the interpreter when the *debugger* option is turned *on*. In order to permit pointers to be dumped using macros, the '$' character is used

User-supplied Macros for Debugging

```
.......
//Macro for dumping a Int8*
#define vm_dump_Int8$(x) fprintf(stderr,"%p",x);
//Macro for dumping a Int16*
#define vm_dump_Int16$(x) fprintf(stderr,"%p",x);
#define vm_dump_Int16(x) fprintf(stderr,"%d",x);
#define vm_dump_Uint8(x) fprintf(stderr,"%u",x);
#define vm_dump_ArrayOfChar(x) fprintf(stderr,"%p",x);
......
```

Sample Interpreter Core with Debugger Code

```
......
iflt:
fprintf(stderr,"iflt\n");
{
    //---------------------------------------
    #define VMLOAD_skip_ fprintf(stderr," IP:read:skip:="); \
    vm_Int32_equals_Address(skip,IPPTR[1]) \
    vm_dump_Int32(skip) \
    fprintf(stderr,"\n"); \
    //---------------------------------------
    Int32 i1;
    Int32 skip;
    fprintf(stderr," SP:pop:i1:=");
    vm_Int32_equals_StackVal32(i1,SPPTR[-1])
    vm_dump_Int32(i1)
    fprintf(stderr,"\n");
    {
        if(i1 < 0)
        {
            VMLOAD_skip_;
            SET_IP(threadedPc+skip);
        }
    }
    IPPTR=IPPTR+2;
    SPPTR=SPPTR-1;
    goto **IPPTR;
}
#undef VM_CURRENT_OPCODE
......
```

Figure 4.33: Interpreter core with debugging code inserted

in substitution for the '*' character. For example, the vm_dump_Int8$ macro is used to dump instances of Int8*.

As can be seen from the iflt example shown in Figure 4.33, the debugger option is compatible with deferred reading and writing. In this example, *skip* may or may not be read in from the instruction stream, depending on the value of *i1*. In order to ensure the debugger only dumps *skip* if, and when, it is read in, Tiger places the code responsible for dumping the value of *skip* directly into the VM_LOAD_skip_ macro.

4.4.4 Profiler

In order to support the profiling of executed instructions and instruction sequences, Tiger provides a profiler option which can be set to on, when required. This profiling is supported by extra code inserted by Tiger into the interpreter core, along with a support library packaged with Tiger. The output of the profiler is a file *profiler.dat* that is written upon interpreter termination. Figure 4.34 illustrates a short section

```
......
From: 0x805802c To: 0x8058040   Count:14952
PUSHAL 5
POSTINC
POP
JUMP 8214
From: 0x8058040 To: 0x8058040   Count:14920
JUMP 8214
From: 0x805804c To: 0x80580b4   Count:1
PUSHAL 5
DEREF
PUSHAL 3
DEREF
GT
JFALSE 8878
PUSHAC 10
PUSHS 8
CALL 2
......
```

Figure 4.34: Profiler Output Using Tiger Profiling Option

from a typical *profiler.dat* file. This file records contiguous sequences of executed instructions and selected operands. The starting address and ending address for each

Global Definitions

```
......
#define VM_EXPORT(_buf,_type,_inst) vm_export_##_type(_buf,_inst)
#include "support/disassembler.h"
#include "support/profiler.h"

#define VM_ON_EXIT VM_PROFILER_EXIT
#define VM_ON_ENTRY VM_DISASSEMBLER_ENTRY\
                        VM_PROFILER_ENTRY

#define VM_THREAD_ON_EXIT
#define VM_THREAD_ON_ENTRY VM_DISASSEMBLER_PER_THREAD_INIT \
                           VM_PROFILER_PER_THREAD_INIT \
......
```

Sample from Interpreter Core with Profiler Code

```
......
#define VM_CURRENT_OPCODE "JFALSE"
#define VM_DISASSEMBLER_GENNAME strcpy(VM_DISASSEMBLER_NAME,"JFALSE"); \
vm_short_equals_char(b,IP[2],IP[1]); \
VM_EXPORT(VM_DISASSEMBLER_NAME+strlen(VM_DISASSEMBLER_NAME),short,b); \

JFALSE:
{
    Int32 a;
    short b;

    vm_short_equals_char(b,IP[2],IP[1]);
    vm_Int32_equals_Int32(a,SP[0]);
    VM_PROFILER_HEADER(IP)
    VM_DISASSEMBLER_HEADER(IP,VM_DISASSEMBLER_GENNAME)
    {
        if(a)
        { ; }
        else
        {
            IP=((mem+b));
            SP+=(1);
            goto *VM_LABELS[IP[0]] ;
        }
    }

    SP+=(1);
    IP+=(3);
    VM_PROFILER_FOOTER
    goto *VM_LABELS[IP[0]];
}
#undef VM_CURRENT_OPCODE
#undef VM_DISASSEMBLER_GENNAME
......
```


Figure 4.35: Support Code for the Profiler

code sequence is recorded, as is the number of times that sequence was executed.

The output of the profiler can be used to select candidate instructions for replication, specialisation (due to the recording of selected operands) and superinstructions[4]. Operands are recorded by the profiler only when the type of the operand is registered with Tiger as being specialisable by using the SPEC modifier (see Section 4.2.1). Thus the profiler only records operands that could form part of a specialised instruction.

An example of the support code generated by Tiger for the profiler can be seen in Figure 4.35. Two files are included to support profiling, namely *profiler.h* and *disassembler.h*. The disassembler code is responsible for the recording of specialisable operands, while the profiler handles everything else. Interpreter and thread initialisation code for profiling is inserted into the VM_ENTRY and VM_THREAD_ON_ENTRY macros and the code responsible for dumping the data collected by the profiler into *profiler.dat* is inserted into the VM_ON_EXIT macro. The most important feature of the code inserted by Tiger into the interpreter core is the use of the VM_PROFILER_HEADER and VM_PROFILER_FOOTER macros to keep track of when an old contiguous sequence of instructions has been completed and a new one is starting. In order to achieve this, Tiger keeps a flag that when set, indicates that a control flow change (i.e. a branch) has occurred. At the start of each instruction, a call to VM_PROFILER_HEADER sets this flag, assuming that a control flow change will occur. At the end of each instruction, a call to VM_PROFILER_FOOTER clears this flag, since a control flow change has not occurred if this point has been reached. Each time a call to VM_PROFILER_HEADER occurs, and before it sets the flow of control flag, it checks this flag and if the flag has not been cleared, it assumes a control flow change has occurred. If this happens, then the old sequence is ended and a new sequence is started (using the argument, IP, as an key for the hash-table of sequences that the profiler stores).

4.5 Conclusion

In this chapter, we have presented the Tiger tool. This tool was critical to the majority of the optimisations implemented in the Java interpreter to facilitate the experimental work presented in this dissertation. Both the various options in Tiger for code genera-

[4]No timing information is recorded by the profiler at present, its primary purpose being to determine which sequences of code are run most frequently.

tion and the underlying support mechanisms have been described in detail. Although the development of Tiger has been with one specific purpose in mind, namely investigation of various Java interpreter optimisations, we believe that many, if not all of the features, will be of use to a number of researchers and students working on interpreter optimisation[5]. In the next chapter we describe the creation of a new Java interpreter core using many of these optimisations, and present a critical analysis of those same optimisations.

[5]Since the completion of this project, the author has completely re-developed the Tiger program to provide all of the features described in this chapter into a freeware tool, vmJen. vmJen uses an XML front-end at present and is packaged with the oc interpreter, along with a set of tools for creating superinstructions, specialisations and replications.

Chapter 5

Construction of an Optimised Java Interpreter

5.1 Introduction

In this chapter[1] we describe our work carried out building a JVM interpreter that is both portable *and* efficient. These two properties do not always go hand in hand, as many JVMs such as Sun Microsystem's HotSpot interpreter engine [Gri98] contain assembly code to optimise it for the particular architecture on which it runs. At the other end of the spectrum, we have the Kaffe Virtual Machine[Wil98], which is highly portable, but not so efficient. We believe however, that portability and efficiency are not necessarily mutually exclusive. By carrying out much of our optimisations at the 'C' source code level we can guarantee a high-level of performance while also maintaining a high level of efficiency.

To demonstrate this, we needed to select a JVM for optimisation. We required this JVM had specific properties, namely:

1. Easy access to interpreter 'C' source code.

2. Good documentation and commenting.

[1]Our early work in constructing an optimised Java interpreter has been published in SAC 2003 [BCGN03]. This early work documents the experimental results of a JVM interpreter the author of this thesis implemented using the vmgen tool. The work documented in this chapter describes the implementation of a JVM interpreter the author implemented using Tiger, and as such represents a superset of the previously published work.

3. Availability for multiple platforms

4. Industrial grade Virtual Machine.

The Virtual Machine we chose for modification was Sun Microsystem's Connected Virtual Machine (CVM) [Sun01]. Although other candidates were considered, such as the Kaffe Virtual Machine and Sun's KVM [Sun00], the CVM stood out as an obvious choice. The entire Kaffe Virtual Machine's runtime engine was too slow while Sun's KVM could not run the full range of benchmarks we deemed necessary to evaluate the performance benefits of our interpreter enhancements[2].

The CVM was designed for embedded devices such as pagers, PDAs and set-top boxes. As it is implemented as an interpreter, it is ideal to run in a compact environment. The memory subsystem has also been designed with embedded devices in mind, with a requirement of 2Mb RAM and about 2.5Mb ROM to be made available to the virtual machine environment [Sun05b]. Some of the main features of the CVM are:

- Optimised Interpreter.
- Fast Java Synchronisation
- Native Thread Support
- Compact and complete memory system
- ROMable Classes
- Small Class Footprint
- 1.3 VM Support
- Java 2 Security support
- Highly portable with a well documented Porting Layer
- Limited Stack Usage

[2]During the course of the work in this thesis, the Sable VM [GH01] has become a more attractive choice, as its development has progressed.

5.2 Customisation Options

The CVM comes with a number of choices for the *profile*, a standard set of features and libraries for the virtual machine. The choices were Mobile Information Device Profile (MIDp), Foundation profile, Personal Profile and RMI profile.

The Foundation profile was chosen, primarily because it was the most compact VM that enabled all our selected benchmarks from Spec98 and Java Grande to be run successfully. The main attributes of this Foundation profile are:

1. A J2SE-based class library.
2. No GUI support
3. A CLDC 1.0 compatibility library

The CVM also comes with a choice of garbage collector, namely *generational*, *marksweep* and *semispace*. The default option, *generational*, worked well and gave reasonable results in our early tests, while the *marksweep* garbage collector was inoperative in the reference implementation. This fact that this collector was broken did not concern us much, since *marksweep* would be expected to be the slowest of all three garbage collection methods. The final method, *semispace*, gave us marginally better performance than *generational* across all our selected benchmarks and remained our choice for all subsequent experimentation.

5.3 Building a Basic Interpreter Core in Tiger

In order to gain access to the code generation options of Tiger, it was necessary to construct a new interpreter core. In order to perform this construction in an incremental manner we used CVM in switch mode and Tiger in switch mode at the same time. The Tiger-generated core was then included next to the CVM core.

We then implemented small groups of instructions in the Tiger interpreter core and removed the corresponding implementations from the original CVM interpreter core. Each time a handful of instructions had been implemented in Tiger and removed from the original CVM core, Tiger was run, generating the new core. Then CVM was rebuilt,

and the interpreter tested on a number of benchmarks to ensure the instructions had been implemented correctly in the Tiger-source.

It was not always evident how to implement instructions in Tiger due to the fact that the actual implementation of instructions in the CVM interpreter core itself was not always clear. This was due to the heavy use of macros in the CVM core. In cases where it was not clear how the macro-expanded instruction might look, we used the GCC 'C' preprocessor to expand the macro definitions for examination.

During the implementation of instructions in Tiger, it was necessary to add the type conversion macros required by Tiger (Section 4.3.2), each time a new type was being read from the stack or instruction stream or a new type was being written to the stack. The original interpreter core also had some fall-through code (for example the implementation of jsr fell-through to the implementation of goto). Instructions such as these were implemented as separate instructions (with no code sharing) in Tiger.

A feature of the old interpreter core that we retained was that some instruction implementations shared the same code (and hence the same label or case statement). For example aload, iload and fload all had the same implementation. This feature was implemented in the Tiger interpreter core for a number of reasons including the fact that it improved interpreter core compactness and also increased the possibility for creation of superinstructions in the bytecode (Section 6.4).

The end result of this stage was a switch-based interpreter generated by Tiger that had the same functionality as the original CVM. At this point we also performed a test by getting Tiger to emit code for a token-threaded interpreter. This token-threaded interpreter passed all correctness tests and performance was more or less equivalent to that of the CVM in token threaded mode. The fact that the performance of our new interpreter core was equivalent to the original CVM was not surprising, since no optimisations had yet been turned on in Tiger.

5.4 Choice of Dispatch Method

The Java Virtual Machine uses a stack-based bytecode to represent the program. Interpreting a bytecode instruction consists of accessing arguments, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction.

aldc_ind_quick in CVM

```
......
        CASE opc_aldc_ind_quick: { /* Indirect String (loaded classes) */
            ObjectICell* strICell = cpGetStringICell(cp, pc[1]);
            ID_icellAssignDirect(ee, &STACK_ICELL(0), strICell);
            UPDATE_PC_AND_TOS_AND_CONTINUE(2, 1);
        }
......
```

aldc_ind_quick in Tiger

```
......
        aldc_ind_quick  SP( - JavaVal32 s1)
                        IP( ObjectICell* oic - next);
        ID_icellAssignDirect_reloc(ee, s1.r,oic);
......
```

Figure 5.1: Translating aldc_ind_quick from CVM to Tiger

Instruction dispatch typically consumes most of the execution time in virtual machine interpreters. The reason is that most VM instructions require only a small amount of computation, such as adding two numbers or loading a number onto the stack, and can be implemented in a few machine code instructions. In contrast, instruction dispatch can require up to 10-12 machine code instructions, and involves a time consuming indirect branch. For this reason, dispatch consumes a large proportion of the running time of most efficient interpreters [EG01].

Section 3.1 introduced a number of dispatch techniques which Tiger supports, namely switch, token threaded, and direct threaded. Of these direct threaded is considered the most efficient, reducing the code to implement instruction dispatch to just three machine instructions on most architectures. It is the most commonly used scheme where interpreter speed is important and is the threading variation we selected for our optimised Java interpreter, Fastcore.

In order to build the proof of concept implementation in a more efficient manner, it was decided to retain the original bytecode and put the translated direct-threaded code into the method descriptor for each method alongside the original bytecode. The direct-threaded code was to be composed of words rather than bytes. Although it would have been possible to have the direct-threaded code stored as bytes, it would have resulted in a huge number of non-aligned reads for the instruction addresses in

the instruction stream. The downside of this word rather than byte approach is that, since the code is more sparse, the instruction cache miss rate increases. However, this is mitigated by the potential for performing various optimisations on the operands of the instructions.

The time chosen to perform the translation for a method was just before a method was run for the first time. In this way we can be sure that the overhead of translating a method is only incurred for methods that will actually be run. To support this Just-In-Time translation, a stub was placed in the threaded code for each Java method. This stub contains a single VM instruction (THREADCODE) which translates the method from bytecode to threaded code.

For the direct-threaded interpreter, each time a method is run, a dispatch takes place to the first instruction address in the direct-threaded code. For a method being run for the very first time, this means a call to the translation stub. This involves a call to the 'C' function responsible for performing the translation. As part of its work, the translation function replaces the initialised direct-threaded code with the direct-threaded version of the bytecode of the method. When the translation function returns, the method is re-launched by dispatching to the first instruction address in the direct-threaded code. This should now be the translated version of the method's bytecode.

5.4.1 Supporting Data Structures

Figure 5.2 shows the addition of the threaded code pointer field to the method descriptor, the initialisation of that field and the THREADCODE instruction definition in Tiger. In Figure 5.2 also shows four other fields that are initialised by the code threading routine. They are:

1. Length of threaded code (threadedLength). This field stores the length of the threaded code (in machine words). The length of the threaded code may be shorter or longer (in terms of words) than the bytecode (in terms of bytes), so the length of the threaded code will often be different than the length of the bytecode.

2. The offsets array (offsets). The position of the program counter is critical to certain operations within the CVM, for example garbage collection and exception

Fields added to the method descriptor

```
.......
struct MethodDescriptor
{
    ....
    Address*     threadedCode;    /* direct-threaded code ptr */
    Uint16       threadedLength;  /* length of direct-threaded code */
    ExceptionHandler*    threadedExceptionTable;
    Int16*       offsets;
    Uint16*      instList;
    ....
};
.......
```

Initialisation of code threading stub

```
.......
        Address THREADCODE_stub[]={ &&THREADCODE };
.......
        MethodDescriptor newMethod;
        newMethod.threadedCode=THREADCODE_stub;
.......
```

The THREADCODE instruction in Tiger

```
.......
        THREADCODE;
                MethodDescriptor* temp=//Get method descriptor...
                /* The threading routine */
                threadCode(temp,methodBlock,constantPool);
                /* Dispatch to threaded code */
                SET_IP((temp->threadedCode));
.......
```

Figure 5.2: Support for direct-threaded code in Tiger

handling. However, because there is not always a one-to-one mapping from the byte-offset of an item in the bytecode and the word-offset of an item in the corresponding threaded code, it is not always easy to determine the position of the true bytecode program counter from the position of the threaded program counter. This situation arises when we start to allow instructions in the threaded code to be a different size than instructions in the bytecode.

A number of solutions to this problem were considered:

- The first was to maintain a bytecode program counter in addition to the threaded program counter, but this was too costly.
- The second option was to perform a deep-modification to the CVM, tracking any places where the original bytecode program counter was required and modifying and dependent data structures (for example stackmaps).
- A third, safer option was to generate the original bytecode program counter any time it was required (which is reasonably infrequently). In order to do this, an array of offsets is created by the code threading function. This array of offsets allows the determination of the bytecode program counter from the threaded program counter. For example, if the threaded pointer offset is i, the bytecode pointer offset is given by i+offsets[i]. Figure 5.3 shows an example of the offsets array. The bytecode is shown over the threaded code equivalent. Note how the new_quick instruction takes up fewer slots in the threaded code than in the bytecode[3]. This causes subsequent instructions in the threaded code to be shifted in relation to the corresponding instructions in the bytecode. The offsets array records these changes.

3. Instruction indices list (instList). Since threaded code does not always have a direct one-to-one mapping to the bytecode, one needs to parse the threaded code directly, when looking for superinstructions. A difficulty in this regard is that the instruction addresses are stored in the threaded code and not the instruction indices. Unfortunately the data structures Tiger supplies for assisting with the parse, rely on the instruction indices. There were a number of solutions to this

[3]This is because a two byte operand in the bytecode is combined into a single 32-bit address and stored as a single word in the threaded code.

problem, but the most efficient was to maintain a list of instruction indices in an array matching the threaded bytecode array. This new array is of the same size as the threaded code array but is composed of unsigned 16-bit integers according to the following rules.

(a) instList[i]=0 if threadedCode[i] contains an operand.

(b) instList[i]=INDEX_OF(*instr*)+1 if threadedCode[i] contains an opcode.

The advantage of this approach is that, when parsing for superinstructions, the location and indices of instructions in the threaded code can be easily determined. Once formed, the instruction index array is unaffected by either replication or superinstruction creation in the method. It is modified, however, when an instruction is quickened or specialised. This behaviour is not arbitrary and is critical to superinstruction parsing.

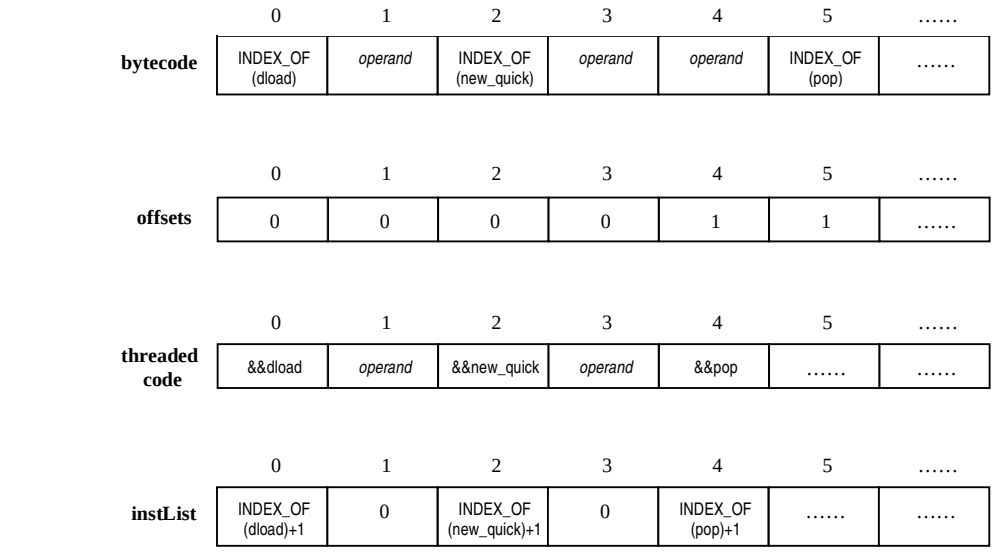


Figure 5.3: Relationship of Bytecode, Threaded Code, Offsets and Instruction Index arrays.

4. Threaded exception table (threadedExceptionTable). Due to the offsets of instructions changing during translation from bytecode to threaded code, exception handling needed some additional support. Typically an entry in the exception

handling table contains a lower bound and upper bound for the bytecode program counter offset, and an offset to jump to if the program counter is in that range when an exception occurs. Since the bytecode program counter was no longer in use, some modifications to this behaviour were necessary. Two options were considered. The first was to translate the threaded program counter offset to the bytecode offset (using the offsets array) each time an exception occurred. Although this would be quite adequate in terms of performance, it was decided to create a threaded version of the exception table, with all the offsets recalculated so that the new table would work with the threaded code offset instead. The extra space required by maintaining this threaded version of the exception table was minimal since most exception tables consist only of a few entries.

5.4.2 Code Translation

Once the appropriate data structures had been put in place for supporting direct threading, the next step was to perform the actual translation work. This work was carried out by the code threading function which took the method descriptor as an argument. All necessary data, such as the bytecode and the bytecode size for the method were available through this method descriptor.

The job of the code threading function was to construct the threaded code from the bytecode. Before any instruction size modifications (Section 5.5) were carried out, there was a one-to-one mapping between bytes in the bytecode and words in the threaded code. This meant that the translation from bytecode to threaded code involved moving through the bytecode, byte by byte. If byte i represented an operand, it was copied into the i^{th} position of the threaded code. If byte i represented an opcode, the address of that instruction is obtained from the labels array VM_LABELS (defined using the labels file generated by Tiger - Section 4.3.4). This address is then written to the i^{th} position of the threaded code.

The net effect of this procedure is that the threaded code and the bytecode have similar contents, except in the threaded code, all instruction indices have been replaced by instruction addresses. At this point in time, there was a direct one-to-one mapping between items in the bytecode and items in the threaded code. It was not until later when we started optimising operands in the direct threaded code and consequently

changing their sizes (Section 5.5) that this one-to-one relationship was violated and the offsets array became necessary. Figure 5.4 shows the results of a simple translation from bytecode to threaded code.

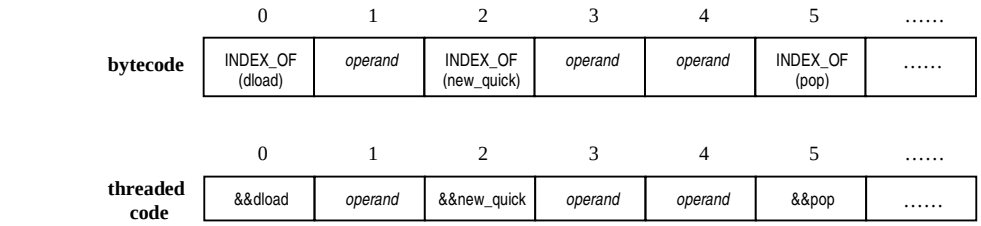


Figure 5.4: Results of Code Threading

To perform this translation efficiently, a token threaded translation routine was incorporated into the code threading routine. The alternative, a switch based translator, would have been substantially slower without necessarily being much easier to implement. The code threading routine consists primarily of a set of labels, one for handling each possible instruction index that can be found in the bytecode. An array of labels (threadLabels) is initialised in the function so that i^{th} element of this array is the label responsible for handling instruction index i. Pseudo code for the translation procedure is shown in Figure 5.5.

5.4.3 Branch Offset Patching

Although not an issue until the one-to-one relationship between bytecode and threaded code is violated (Section 5.5), we describe the mechanism for fixing jump offsets in the threaded code here. When this one-to-one mapping is gone, there is no longer any guarantee that the target for a branch or jump in the threaded code is the same as in the bytecode. The solution taken here was to mark all instructions in the threaded code that contained jump offsets (e.g. iflt, goto) and perform a second pass once the whole threading process had completed. Since the offsets array is constructed during the threading process, this is available to help in the recalculation of offsets in the second pass.

On the second pass through the threaded code, each time an offset (whether relative or absolute) is found, the translation routine determines where the target of the offset is

```
    void* threadLabels[]={
                            &&thread_nop,
                            &&thread_aconst_null,
                            &&thread_iconst_m1,
                            ......
                          };
    int i=0;

    while(i<byteCodeSize)
    {
        goto *threadLabels[byteCode[i]];

        thread_nop:
            threadedCode[i]=&&nop;
            /* No operands */
            i++;
            continue;

        thread_aconst_null:
            threadedCode[i]=&&aconst_null;
            /* No operands */
            i++;
            continue;

        thread_iconst_m1:
            threadedCode[i]=&&iconst_m1;
            /* One operand */
            threadedCode[i+1]=byteCode[i+1];
            i=i+2;
            continue;

        ......
    }

......
```

Figure 5.5: Pseudo-code for Code Threading Procedure

in the bytecode. The old bytecode offset for the branch is used to find the target of the branch in the bytecode. Then, by examining the offsets array, it can be determined how much the branch source and branch target have moved in relation to each other during the threading process. This figure is added to the branch offset in the threaded code, thus correcting the branch. Figure 5.6 illustrates pseudo-code for this offset-correction procedure.

While the actual branch offset patching procedure is a bit more complicated (due to different types of offset and the lookupswitch and tableswitch instructions), the actual algorithm used is quite similar to that presented in the example.

```
......
char branch[]=
{
    /* Initialised during threading */
    /* branch[i]=1 if threadedCode[i] contains an offset */
    /* else branch[i]=0 */
};

Uint16 offsets[]=
{
   /* Initialised during threading */
};

int i=0;
while(i<byteCodeLength)
{
   if(branch[i]==1)
   {
       int offset=threadedCode[i];        // The branch to be fixed

       int byteSource=i+offsets[i];
       int byteTarget=byteSource+offset;

       // Calculate how much threaded offset need to be adjusted by
       int delta=offsets[byteSource]-offsets[byteTarget];

       threadedCode[i]+=delta;
   }
}
......
```

Figure 5.6: Branch Offset Patching

5.4.4 Quickable Opcodes

The threading process does much of the work required for getting bytecode translated into direct-threaded code. *Quickable* instructions present a particular challenge to this process though. These quickable instructions are instructions that can be replaced by faster, more optimised instructions, but only after (usually immediately after) they are executed for the first time. Thus, the replacement of a quickable instruction occurs essentially in a Just-In-Time manner. This replacement process is called *quickening* and the instruction replacing the quickable instruction is called a *quick* instruction. Typically a quick instruction will perform essentially the same work as the quickable instruction it replaces, minus some checks that are only necessary the first time the instruction is executed.

Quickable instructions, even if they are translated to their quickable threaded equivalents at translation time, will need to undergo a further translation if and when they are quickened (i.e. executed for the first time). Quickable instructions in the instruction stream presented so many difficulties, particularly with superinstructions (see Section 6.4.4 for a discussion), that we experimented with prematurely quickening instructions during translation to direct-threaded code. Unsurprisingly this broke the virtual machine. The main cause of this was that sometimes, in order to determine what some non-quick instructions were ultimately going to be quickened to, it was necessary to run static initialisers for a class. Thus sections of code were being executed long before they should normally be.

Having decided against premature quickening, it was necessary to modify the quickening routines to ensure they wrote the correct threaded code for instructions that had been quickened. The first step to support direct-threaded quickening is performed during translation. It was necessary to store information such as the index of the quickable instruction. In order to do this, a structure NonQuickInfo was created specifically for this purpose. As each quickable instruction had at least one operand, this operand was absorbed into the NonQuickInfo structure and then the address of the NonQuickInfo structure was written over the operand. Finally, instead of writing the instruction address for a dedicated quickable instruction, a generalised quickable instruction, QUICKEN_OPCODE was written to the instruction stream. Figure 5.7 shows an example of this, where the quickable instruction `new` is replaced by the instruction

QUICKEN_OPCODE in the threaded code.

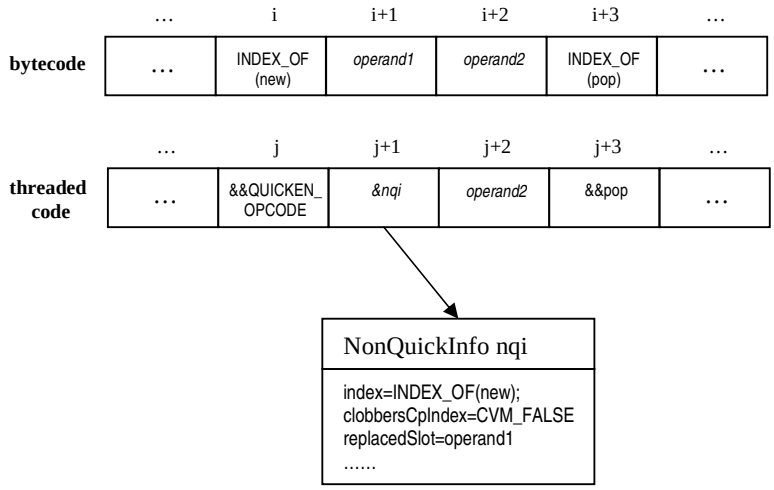


Figure 5.7: Threading for Non-Quick Instructions in Fastcore

Later, during execution of the method, if the QUICKEN_OPCODE instruction is encountered, this means that it is time for the instruction it replaced to be quickened. The QUICKEN_OPCODE instruction retrieves the NonQuickInfo structure through the pointer (its first operand). Once it has access to the structure, it retrieves the original operand from inside the structure and restores it, overwriting the pointer to the NonQuickInfo structure in the instruction stream. It then retrieves all the other information required for quickening and then calls the quickening routine. Upon completion, the quickening routine writes the address for the appropriate instruction over the address for the QUICKEN_OPCODE instruction in the instruction stream. It also writes any modified operands to the instruction stream. Finally, when the quickening code is writing the address for quickened opcode *instr* at offset i, it also updates the instList array (Section 5.4.1) by setting instList[i]=INDEX_OF[*instr*]+1. This is essential to the static superinstruction parsing process.

There were alternative approaches to quickening than to use a single quickable instruction (QUICKEN_OPCODE) in combination with the NonQuickInfo structure. For example, it would have been possible to have tailored quickable instruction code for each of the possible quickable instructions. However, the approach taken was deemed to be more flexible and slimmed down the interpreter core a little, due to only one

quickable instruction being implemented there. This NonQuickInfo structure proved to be useful again later, when dynamic replication was being implemented (Section 7.4.1).

5.4.5 Threaded Exception Handler

The last significant task in the bytecode threading function was to construct the threaded exception table. For each method, this threaded exception table has identical structure and size of the standard bytecode exception handler with the exception of the code offsets. Each row in an exception handler consists of a lower bound (startpc), an upper bound (endpc), an exception type (catchtype) and offset of the code for handling the exception (handlerpc). If an exception of catchtype occurs when the program counter is between the upper bound and the lower bound, then the program counter is set to the exception handler, handlerpc. In order to convert the exception handling table of a method to a threaded version, all that is required is to move through the table, row by row, changing offsets for startpc, endpc and handlerpc. These offsets only need to be changed due to the possibility of instruction offsets changing during the translation from bytecode to threaded code. Once more, the offsets array, produced by the threading process, is used. Figure 5.8 illustrates the code required to create the threaded exception table.

5.4.6 Stackmaps and Garbage Collection

After translation to threaded code, the dispensing of the bytecode pointer and the violation of the one-to-one mapping between bytecode and threaded code, the garbage collection process was effectively broken. The two solutions to this problem were either to recalculate the stackmap for the threaded code or to map the threaded program counter back to a bytecode counter whenever the stackmap entry for the current program counter was required. In the end, the latter approach was chosen, as the overhead of threaded program to bytecode program conversion was minimal and relatively infrequent. This conversion took place, as with previously described program counter conversions, by using the offsets array stored in the method descriptor.

```
......

ExceptionHandler* teh;
ExceptionHandler eh;
JavaMethodDescriptor* jmd;

jmd= /* The method descriptor for the method being translated */;

eh= jmdExceptionTable(jmd);                    //Get bytecode exception table
ehEnd= eh + jmdExceptionTableLength(jmd);      //Get bytecode exception table length

/* Create enough space for new threaded exception table */
teh=(ExceptionHandler*) malloc(sizeof(ExceptionHandler)* jmdExceptionTableLength(jmd));

/* Store the new threaded exception table in the method descriptor */
jmd->threadedExceptionTable=teh;

/* Patch up all the rows in the exception table */
for (; eh < ehEnd; eh++)
{
   teh->startpc=eh->startpc-offsets[eh->startpc];
   teh->endpc=eh->endpc-offsets[eh->endpc];
   teh->handlerpc=eh->handlerpc-offsets[eh->handlerpc];
   teh->catchtype=eh->catchtype;
   teh++;
}

......
```

Figure 5.8: Threaded Exception Table Creation in the Fastcore interpreter

5.5 Initial Optimisations

Once the interpreter had been imported into Tiger successfully, a number of optimisations were applied to the code. Individually, many of these optimisations had only a minimal effect, but cumulatively the increases in performance were substantial. Performance results for the optimised VM are presented at the end of this section.

5.5.1 Multiple Dispatches for Conditional Branches

In the original interpreter, conditional instructions have a single dispatch shared regardless of whether the branch is taken. If the single dispatch is split into two, one for when the branch occurs and one for when the fall-through path is taken, the overall branch misprediction rate is improved considerably. Figure 5.9 shows how the Tiger definition for the iflt instruction looks before and after this optimisation. In both examples, there is an implicit dispatch added to the end of the instruction by Tiger. Additionally in the multiple dispatch example, the SET_IP macro ends in a dispatch.

5.5.2 Operand Modification

A number of modifications to the instruction stream were possible after conversion to threaded code. This possibility is attributed to the fact that each byte that represents an operand in the bytecode is copied into a 32-bit word in the threaded code. In order to make the most of this extra space, the first optimisation we applied was to convert multiple byte values in the bytecode into single word values in the threaded code. For example, the bytecode contains a large number of 16-bit numbers spread over two bytes. During conversion to threaded code, the 16-bit number is reconstituted and stored in a single word in the threaded code. Figure 5.10 shows the translation of the invokesuper_quick instruction during the threading process. In the example, a 16-bit number is retrieved from two byte-sized slots in the bytecode using the GET_INDEX macro (defined elsewhere). This number is then stored in a single slot (word) of the threaded instruction stream. When the translation is finished, the bytecode pointer is incremented by three bytes (the opcode index and two bytes for the number) while the threaded pointer is only incremented by two words (opcode address and one word for the number). Note that, because of the new operand layout, some minor modifications

```
Single Dispatch:
                iflt SP( Int32 i1 - )
                     IP( Int32 skip - next);

                if(i1 < 0)
                {
                        IPPTR=IPPTR+skip;
                }
                //Implicit dispatch at end of instr
                ----------------------------------------
```

```
Extra Dispatch:
                iflt SP(  Int32 i1 - )
                     IP( Int32 skip -  next);

                if(i1 < 0)
                {
                        //The SET_IP macro contains a dispatch
                        SET_IP(IPPTR+skip);
                }
                //Implicit dispatch at end of instr
                ----------------------------------------
```

Figure 5.9: Adding an extra dispatch to the iflt instruction

to the affected instructions are required in the interpreter core. This type of optimisation was the first to break the one-to-one mapping between items in the bytecode and items in the threaded code.

```
......
thread_invokesuper_quick:
{
        //Store the opcode address in the threaded code
        threadedCode[threadedPtr]=&&invokesuper_quick;

        //The GET_INDEX macro gets a 2-byte number from the bytecode
        threadedCode[threadedPtr+1]=GET_INDEX(bytecodePtr+1);

        //Note the difference in the pointer increments
        threadedPtr+=2;
        bytecodePtr+=3;

        //Continue translation
        THREAD_NEXT;
}
......
```

Figure 5.10: Operand combining

5.5.3 Constant Pool Inlining

Quite a number of instructions in Java access items in the constant pool. The motivation behind the constant pool is primarily one of code compaction. Rather than storing a large item as an operand in the bytecode, the large item could be placed in the constant pool and an index to the item can be placed in the instruction stream. While this does have the advantage of code compaction, it increases the overhead each time an instruction needs to fetch an item from the constant pool, as it must first get the index from the instruction stream and then look up the appropriate entry in the constant pool.

After the translation to threaded code, there was some spare capacity in the instruction stream. For this reason, it was decided to inline any 32-bit sized items from the constant pool into the threaded code. This meant that there was no code expansion since we were only using the spare capacity (and that of the eliminated operand

holding the index to the item in the constant pool). For any items larger than 32 bits we performed a compromise procedure by storing the address of the item in the constant pool into the instruction stream, rather than using an index. This resulted in slightly faster constant retrieval times than the index to constant pool approach. More importantly, this removed the need for a local pointer variable for the constant pool in the main interpreter function. Since local variables are often stored in registers, this in turn reduced the pressure on registers in our interpreter loop, allowing us to use those registers in a more optimal manner (Section 5.5.9). Figure 5.11 shows the translation for the new_quick instruction. Normally this instruction would expect an index for the constant pool that holds the address of a class block. However, the translation procedure for the new_quick instruction retrieves the address of the class block and stores it directly into the threaded instruction stream as an argument for the new_quick instruction. As with operand combining above, this type of optimisation entails some minor modifications to the interpreter core. In this case the modification is a simplification since the code required to do a constant-pool retrieval is no longer necessary.

5.5.4 Conditional Loading of Operands

In the original version of our interpreter, many instructions loaded their operands, whether or not they were required. Most notable were the conditional instructions which loaded branch offsets from the instruction stream regardless of the result of the condition that was being tested. All the instructions where this situation existed were rewritten so that operands were only loaded if and when they were required. The deferred loading capability of Tiger (Section 4.2.7) made this possible. Figure 5.12 shows the ifgt instruction before and after the application of this optimisation. Note how, after the optimisation, the variable skip is only loaded if the condition is true (by using the Tiger defined macro VMLOAD_skip_). Unlike the previously discussed optimisations, this particular type of optimisation works equally well for switch, token threaded and direct threaded dispatch interpreters.

```
......
thread_new_quick:
{
       Uint16 index;

       //Store the opcode address in the threaded code
       threadedCode[threadedPtr]=&&new_quick;

       //The GET_INDEX macro gets a 2-byte number from the bytecode
       index=GET_INDEX(bytecodePtr+1);

       //The cpGetCb macro takes the constant pool address and an index...
       //...into the constant pool. It returns the entry at that address.
       threadedCode[threadedPtr+1]=cpGetCb(constantPool,index);

       //The pointer increments
       threadedPtr+=2;
       bytecodePtr+=3;

       //Continue translation
       THREAD_NEXT;
}
......
```

Figure 5.11: Constant Pool Inlining

```
Before Optimisation:

                ifgt SP( Int32 i1 - )
                     IP( Int32 skip - next);

                if(i1 > 0)
                {
                        SET_IP(IPPTR+skip);
                }
                ----------------------------------------
```

```
After Optimisation:

                ifgt SP(  Int32 i1 - )
                     IP( +DEFER Int32 skip - +DEFER next);

                if(i1 > 0)
                {
                        VMLOAD_skip_;
                        SET_IP(IPPTR+skip);
                }
                VMLOADnext;
                ----------------------------------------
```

Figure 5.12: Conditional Loading of Operands

5.5.5 Redundant Stack Push Elimination

As outlined in Section 4.2.5, it is possible to eliminate redundant stack updates using Tiger. For our interpreter, two instructions contain redundant pushes, namely dup, which duplicates the topmost item on the stack and dup2 which duplicates the two topmost items on the stack. By using the +DEFER modifier to prevent Tiger adding code to perform the redundant pushes, the implementation of these instructions can be made more efficient. Figure 5.13 shows the Tiger source and generated code for the dup2 instruction with the redundant stack pushes eliminated. Note how only two non-redundant pushes occur in the generated code.

5.5.6 Synchronised Method Instructions

In Java, a block of code can obtain three kinds of lock:

1. An instance lock, associated with a particular object.
2. A static lock, associated with a particular class.
3. No lock.

If a method is *synchronised*, it is associated with an instance (object) lock or a static (class) lock. In either case, upon invocation, the execution of such a method will block until the lock can be acquired. When the method completes the lock must be released. Thus there is some synchronisation overhead when invoking methods and also when returning from them.

Before the invocation of each Java method, the original interpreter checked to see if the method was synchronised or not. Although this check was not too costly, it was rendered unnecessary through the introduction of a new instruction, sync which performed all the work required to synchronise a newly launching method. Handling the return from a synchronised method was dealt with in a similar manner. Returning from a method was only possible through two instructions, dreturn and return. Two additional instructions, dreturn_sync and return_sync were created. These new instructions contained all the work required to handle the return from a synchronised method. The main benefit of these new instructions is that they gave a slight performance improvement and simplified the method invocation and return code in the

```
Tiger Source:
......
        dup2 SP( JavaVal32 s1 JavaVal32 s2
                - JavaVal32 s1,JavaVal32 s2,JavaVal32 s1,JavaVal32 s2)
            IP( - next);
            /* Duplicate the top item on the stack */
            ------------------------------------------------
......
```

```
Tiger Generated Code:
......
        dup2:
        {
            JavaVal32 s1;
            JavaVal32 s2;
            vm_JavaVal32_equals_StackVal32(s2,SPPTR[-1])
            vm_JavaVal32_equals_StackVal32(s1,SPPTR[-2])
            {
                /* Duplicate the top 2 items on the stack */
                vm_StackVal32_equals_JavaVal32(SPPTR[0],s1);
                vm_StackVal32_equals_JavaVal32(SPPTR[1],s2);
            }
            IPPTR=IPPTR+1;
            SPPTR=SPPTR+2;
            goto **IPPTR;
        }
......
```

Figure 5.13: Optimised dup2

interpreter function. Figure 5.14 shows the translation of the dreturn instruction. Note how the current method in translation is tested to see if it is synchronised. If it is, the dreturn_sync instruction is used. If not, the normal dreturn instruction is used instead. As with the previous optimisation, this particular type of optimisation works equally well for switch, token threaded and direct threaded dispatch interpreters (although the translation process looks slightly different).

```
......
thread_dreturn:
{
        if(mbIsSynchronised(currentMethod))
        {
            threadedCode[threadedPtr]=&&dreturn_sync;
        }
        else
        {
            threadedCode[threadedPtr]=&&dreturn;
        }

        //The pointer increments
        threadedPtr+=1;
        bytecodePtr+=1;

        //Continue translation
        THREAD_NEXT;
}
......
```

Figure 5.14: Translation of dreturn

5.5.7 Faster Java Method Dispatch and Return

There are a number of different types of method in the interpreter. They are Java methods, JNI methods, CNI methods, abstract methods and miranda methods[4]. Of all of these methods, the Java methods, the standard interpreted method is by far the most common. Table 5.1 shows counts for the various method types encountered for selected benchmarks from the SPECjava98 suite. *Lazy* methods are JNI methods from

[4]A Miranda method is an automatically created method. In the case of CVM Fastcore, these methods are used to deal with missing interface methods and cases where a non-public method shares a name with an interface method.

dynamically loaded classes that must be resolved the first time they are invoked. Once resolved, these methods are invoked as per usual and thus are included in the JNI count in addition to the Lazy count. Note from the table how Java methods dominate all other types of method (representing 98.88% of all types of method). The next most common type of method is JNI (representing 1.12% of all types of method).

Type	Jack	Mpeg	Compress	Javac	Jess	Db	Mtrt	%
Java	49,568,419	108,508,878	225,997,430	96,391,709	110,983,653	117,322,713	286,117,020	98.88
JNI	2,797,053	1,253,789	8,434	4,233,584	2,675,315	82,230	200,034	1.12
CNI	2,659	2,091	1,285	6,536	4,497	982	1,517	0.00
Lazy	5	5	5	5	5	5	5	0.00

Table 5.1: Method types encountered in the SPECjava98 suite.

Normally when invoking a new method from the interpreter, the invocation opcode jumps to a label callmethod which is shared by all of the invocation opcodes. The code at this callmethod label examines the type of method being invoked and performs all the initialisation work required before dispatching to that method.

It was decided to streamline the interpreter by moving the invocation code for Java methods into the actual invocation instructions themselves. If the invocation instructions find that the method to be invoked is not a Java method, they still jump to the callmethod label, as before. The major benefit here is that a single dispatch point at the end of the callmethod label's code is no longer being used for all the Java methods. Instead, the dispatch point at the end of the invoking instruction is being used. This has the effect of reducing the overall branch misprediction rate somewhat.

A similar approach to reducing mispredictions was taken with the instructions returning from a method. Typically the return instruction would perform minimal work before jumping to a common label for all return instructions. The code at this label handled most of the work of the return and then dispatched to the appropriate instruction. Once again, we had a dispatch that was shared among several return instructions such as dreturn, areturn and return. In order to increase branch prediction accuracy, we removed the shared code for handling returns from methods and gave each instruction its own copy of the return handling and dispatch code.

5.5.8 Software Barrel Shifting

A barrel shifter has been incorporated into most 32-bit processors in use today. This barrel shifter enables left/right shift and rotate operations to be completed in one clock cycle and is essential for the speed of operations such as multiplication and table lookups. Despite having incorporated fast barrel shifters into their previous Pentium processors, along with their 486 and 386 processors, Intel removed the barrel shifter from their Pentium 4. Thus a typical shift operation, that on a previous processor would have taken only one cycle, now takes 4 to 6 cycles on a Pentium 4.

During translation of the bytecode, the opportunity exists to alleviate some of the speed losses due to the lack of a barrel shifter. Consider for example the fload instruction which has an operand representing the index of the local variable that it is supposed to place onto the stack. At some stage the fload instruction will access locals[*operand*]. This is equivalent to accessing the item stored at memory location locals+(*operand**sizeof(LocalVarType)) where LocalVarType is the type of the local variables array. Unfortunately, this operation is slower on a Pentium 4 due to the lack of a barrel shifter. Instead of incurring this overhead each time fload is executed, it is better to incur it just once, at translation time. Thus, during translation, the *operand* to fload is replaced with *operand**sizeof(LocalVarType), negating the need for a costly multiplication later. When the fload instruction is retrieving the local variable using this new pre-shifted operand, one must be careful to avoid simply adding the new pre-shifted operand to the base of the locals array, since the 'C' compiler will interpret this as pointer arithmetic and will perform a multiplication on the already shifted operand. Figure 5.15 shows the technique for pre-shifting at the translation stage and also shows the correct way to use the pre-shifted operand in the implementation of the fload instruction in the interpreter core. This technique is only used with direct-threaded dispatch since the pre-shifted operands take up more space in the instruction stream. In direct-threaded dispatch this extra capacity is already available. In the other methods of dispatch, the bytecode would need to be expanded, most likely wiping out any possible benefits of pre-shifting.

Pre-Shifting at Translation Time

```
......
        thread_fload:
        {
                    //Store the address for the instruction
                    threadedCode[threadedPc]=&&fload;

                    //Pre-shift the operand
                    threadedCode[threadedPc+1)=bytecodePc[1]*sizeof(SlotVal32);

                    //Increment pointers
                    threadedPc+=2;
                    bytecodePc+=2;

                    //Translate next instruction
                    THREAD_NEXT;
        }
......
```

Implementation of fload in the Interpreter Core

```
......
        fload   SP( - SlotVal32 f1 )
                IP( Uint8 i1 -  next);

                //Must convert locals to a byte-sized pointer...
                //...before addition of pre-shifted operand
                f1=*((SlotVal32*) (((char*) locals)+i1));
        ------------------------------------------------
......
```

Figure 5.15: Pre-Shifting Operands

5.5.9 Optimising Register Allocation

In order to speed up the interpreter, it is desirable to hold some of the more heavily used variables such as the stack pointer and instruction pointers in registers. Simply declaring them as local variables can get the variables held in registers, but not as often as one would wish. Firstly, the register allocator in the compiler decides which items get stored in registers and which do not. Secondly, even if a register is allocated to a particular part of the code there is no guarantee that a register will be allocated to the same variable later in the same function.

In order to ensure that as many registers are available for the local variables that really count, it was decided to limit the number of local variables to the bare minimum. Strategies such as Constant-Pool Inlining (Section 5.5.3) and removing little-used local variables did improve performance but not greatly. Where it was not possible to remove a local variable entirely, its scope was reduced to a minimal level.

The register allocator in `GCC` has a difficult time optimising the interpreter core's code due to its non-contiguous nature. It was possible to assist the `GCC` register allocation routine with the task, however. After local-variable removal from the interpreter core, it became possible to allocate registers explicitly to the important local variables that were left behind[5].

In order to force `GCC` to keep a specific variable in a register, it is not sufficient to use the `register` keyword. One must also specify the actual register to be used by using the `asm()` directive. Figure 5.16 shows an example of how this is done in the interpreter on the x86 platform[6], with the threaded program counter being allocated to the `bx` register. This explicit register allocation is essentially guesswork since, by explicitly allocating registers, one is reducing the number of registers the compiler's register allocator has to work with. After experimentation, it was found that allocating a register to the threaded program counter was optimal. Allocating a dedicated register to the stack pointer degraded performance a little and allocating other registers to the stack pointer resulted in register spill errors during compilation. Later, as the

[5]When explicit register allocation is attempted while register pressure is too high, the compile will fail with a register spill error when the compiler finds it cannot keep the requested variables in the specified registers. This was invariably the case when explicit register allocation was attempted before we started to eliminate local variables from the interpreter loop.

[6]Platforms based on x86 compatible CPUs from companies such as Cyrix, Intel and AMD.

```
......
register Address*  threadedPc asm("%ebx");
......
```

Figure 5.16: Declaration of Register Variable

interpreter was more heavily modified, it was found that explicitly allocating registers to the threaded program counter was sometimes decreasing performance. This illustrates nicely the unpredictability of register allocation. The approach that we have taken is to construct two versions of the interpreter at each stage, one with explicit register allocation, and one without. Profiling each of the running virtual machines yields useful data, not only in terms of time but also in terms of loads, stores and instruction cache misses. Typically the better (i.e. faster) interpreter has fewer loads, stores and instruction cache misses due to better register allocation.

It must be stressed that explicit register allocation not only yields unpredictable results but is also architecture dependant. On the other hand, sometimes the gains are considerable, and the additional porting work required when moving to another platform is negligible.

5.6 Discarded Optimisations

Not all of the optimisations that were tested proved to be effective. One such optimisation, Speculative Java Method Invocation was implemented but proved to be a failure. Essentially it involved invoking every method as if it were a Java method. Non-Java methods would contain a single Java instruction NONJAVA_METHOD which would undo all the speculative invocation work and re-do the invocation, correctly this time. Unfortunately, the work required to undo the speculative invocation was too great and incurred too frequently. As a result, this optimisation actually slowed the interpreter somewhat.

Another optimisation which was attempted was Absolute Addressing. In this optimisation, the offsets for branches in conditional instructions were replaced by the actual memory address of the branch targets. When the branch was to occur, a dispatch to that address would take place. Unfortunately, the measured performance benefits were

minimal, even on the Pentium 4 without the barrel shifter. After tests, it was decided to dispense with this particular optimisation because it would have reduced the scope for instruction specialisation without necessarily giving much of a benefit.

Finally, another optimisation which we examined was optimised Forward Branch Instructions. Normally on a branch, the interpreter performs a check to see if the branch is a backwards branch. If so, it checks if garbage collection needs to run. If it does, extra code needs to be run to spill variables such as the stack pointer from local variables back to memory. To save the cost of the backwards branch check, two sets of branch instructions were added to replace the previous branch instructions. One set was used for forward branches where no check was required and another set was used for backward branches. Thus for the iflt instruction, there was now two versions, iflt_forward and iflt_backward. The determination of which version of instruction was performed at translation time. Figure 5.17 shows an example of how the translation for iflt was carried out. Unfortunately the results for this optimisation were inconsistent at best and at worst slowed the interpreter. Due to the debatable nature of this optimisation it was later removed. During our research on static replication we realised that adding forward branch instructions was akin to adding small numbers of replicated instructions. For reasons discussed in Chapter 6, this can frequently degrade the performance of the interpreter.

5.7 Experimental Results on the Optimised Interpreter

The importation of CVM into Tiger and the optimisation of the interpreter core was with a specific purpose in mind: to prove that interpreters can be made much faster by applying modern compiler optimisation techniques to them, without the need to resort to assembly language. Interpreters optimised in this way tend to be much more portable as the optimisations are all applied at the 'C' source level.

To test our thesis we compared our new optimised interpreter against a number of other small JVMs using the SPECjvm98 (Table 5.2) and Java Grande (Table 5.3)

```
......
thread_iflt:
{
       //Get the offset for the branch from the bytecode
       Int16 offset=getInt16(bytecodePtr+1);

       //Which version of the branch should we use?
       if(offset<=0)
       {
           threadedCode[threadedPtr]=&&iflt_backwards;
       }
       else
       {
           threadedCode[threadedPtr]=&&iflt_forwards;
       }

       //Store the offset in the threaded code
       threadedCode[threadedPtr+1]=offset;

       //Update pointers
       bytecodePtr+=3;
       threadedPtr+=2;

       //Continue translation
       THREAD_NEXT;
}
......
```

Figure 5.17: Forward Branch Introduction at Translation Time

benchmarks to examine performance[7]. These benchmarks consist of several large programs with real data, which are intended to be representative of a wide range of Java applications.

Program	Description
_201_compress	modified Lempel-Ziv compession
_202_jess	Java Expert Shell System
_209_db	small database program
_213_javac	compiles 225,000 lines of code
_222_mpegaudio	an MPEG Layer-3 audio stream decoder
_227_mtrt	multithreaded ray-tracing program
_228_jack	a parser generator with lexical analysis

Table 5.2: Spec98 Benchmark programs used to evaluate VM performance

Program	Description
euler	computational fluid dynamics
moldyn	molecular dynamics simulation
monte	Monte Carlo simulation
raytrace	3D ray tracer
alpha	alpha-beta pruned search

Table 5.3: JavaGrande Benchmark programs used to evaluate VM performance

Using these benchmarks, we also examined the performance of the original unoptimised interpreter in token threaded mode, its fastest possible configuration out of the box. Although it would have been possible to run the original interpreter in switch mode, our experiments showed it to be 17% to 52% slower. As a result, we do not present results for the switch-based interpreter here. In the case of the original unoptimised interpreter and our direct-threaded optimised interpreter, we used the semi-space garbage collector rather than the generational one due to the 1% to 3% speed improvement it offered.

[7]In later testing, we found it more practical to resort to fewer benchmarks for code maintenance reasons. We found that the Spec98 benchmarks were sufficient and, more importantly, were more acceptable as a broad measure of performance than the JavaGrande suite. Previous work [BDGW02] has found that the JavaGrande benchmarks spend a considerably greater amount of time in program methods, which, as we enhanced our VM interpreter would have likely resulted in overly optimistic speedups (due to the fact that program method execution speed is closely tied to dispatch speed).

For comparison, we also measured the speed of the widely used Kaffe JVM. Kaffe is a freely available, robust, highly portable JVM which is available under the GNU General Public License. A commercial version of Kaffe is sold for use in embedded systems. It is important to note that the results we present are for the public version not the commercial one, although the two versions share much code. With Kaffe, we tested both the interpreter and the JIT compiler. The results show what can be expected from a simple, unoptimised interpreter, and a small, portable JIT compiler. Finally, we measured the performance of Sun Microsystem's desktop implementation of Java 2 Standard Edition (J2SE) interpreter from the Hotspot Client VM. Hotspot uses a sophisticated interpreter [Gri98], coded in hand-tuned assembly language. In addition to careful assembly language programming, it also uses a number of optimisations, such as combining common sequences of bytecode instructions into superinstructions, and processor specific (x86 CPU) optimisations for floating point operations. It also stores the topmost elements of the stack in registers and uses a complicated stack-caching system for managing the various states of the stack. To avoid code explosion due to stack caching states, the machine code of the interpreter is generated in memory at run time. The result is that, although the Hotspot interpreter is fast, it is complicated and quite unportable.

Figure 5.18 shows the running times of the benchmarks running on each of the implementations of the JVM, relative to our interpreter (Fastcore, whose speed is represented as 1). The most striking result is for the Kaffe interpreter which is, on average, 5.76 times slower than our interpreter. The Kaffe interpreter is not at all optimised. In particular, it resolves method names and constant pool references every time they are used, rather than just once - the first time they are used. The Kaffe interpreter demonstrates very well that it is easy to write a very inefficient interpreter.

The original CVM interpreter is an average of 31% slower than our optimised interpreter. It does particularly well on *db*, where it is only 16% slower. We investigated the reason for this variance by profiling the code. We found that in the original CVM, only 87% of the time for the *db* benchmark is spent in the interpreter. The rest of the time is spent in the run time system, on garbage collection, synchronisation and native methods. In contrast 98.74% of the time for *mpeg* and 99.89% of the time for compress is spent in the interpreter. So, although the speedup in the interpreter core is similar across all programs, the overall speedup for *db* is lower, since there is no change in the

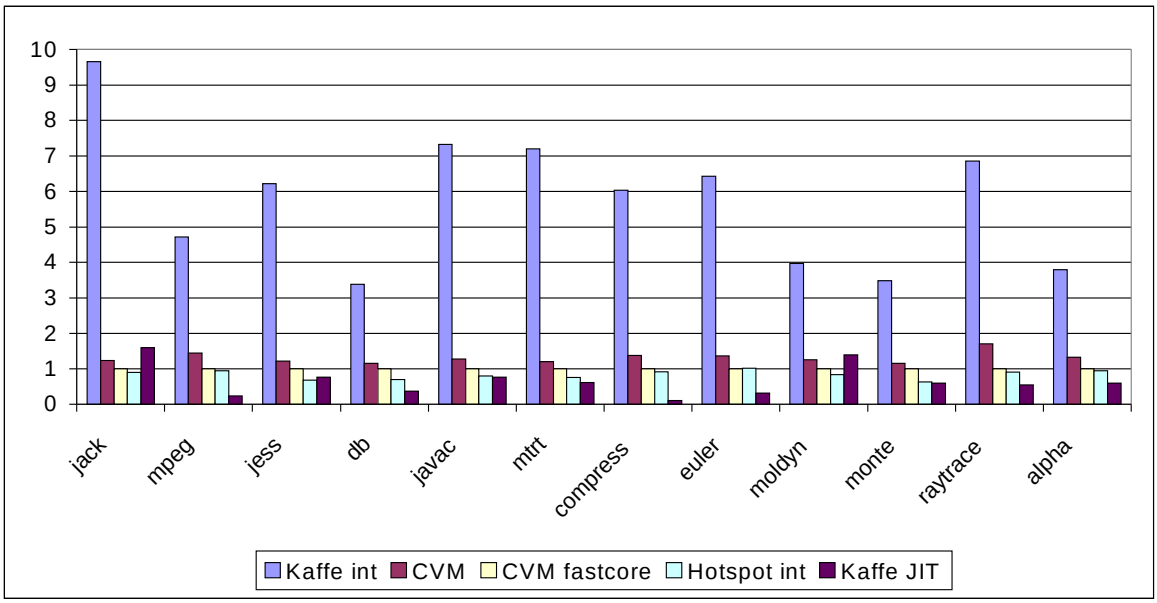


Figure 5.18: Benchmark running times on various JVMs relative to our interpreter (Fastcore)

execution time of the run time system. For this reason, programs such as *compress* (38% faster), *mpeg* (44% faster) and *raytrace* (71% faster) give a better indication of the relative speeds of the interpreter cores.

The Hotspot interpreter is on average 20.4% faster than our Fastcore interpreter. There are two main reasons for this. Firstly, Hotspot has a much faster run time system than CVM. This can be seen especially strongly in the *db* benchmark, which runs 34% faster on Hotspot. The Hotspot runtime system is large and sophisticated, and would not be suitable for an embedded system. Furthermore, much effort has been put into tuning the Hotspot run time system as it is used more widely than CVM. The second reason that Hotspot outperforms our Fastcore interpreter is that the Hotspot interpreter is faster than our interpreter. Its dynamically-generated, highly-tuned assembly language interpreter is able to execute bytecodes more quickly than our portable interpreter written in 'C'. The difference in speeds of the interpreter core can be seen by examining the benchmarks that spend most of their time in the interpreter core: *compress* is 9.1% faster and *mpeg* is 5.2% faster on the Hotspot interpreter. Our

interpreter is actually a little (1.9%) faster on *euler*.

Finally, the Kaffe just-in-time (JIT) compiler is, on average, more than twice as fast as our Fastcore VM. In fact, the results for *mpeg* and *compress* demonstrate that it is four to eight times faster at executing bytecodes than our interpreter. On other benchmarks, its poor run time system slows it down to the extent that it is actually substantially slower on the *jack* benchmark. This demonstrates clearly that a JIT compiler does not always guarantee better performance than an interpreter. The run time system must also be considered. It should also be noted is that the Kaffe JIT compiler does not produce especially fast code. In particular, the mixed-mode Hotspot compiler/interpreter for desktop machines is usually more than twice as fast. However, the Kaffe JIT compiler is simple, and similar to the commercial version which is used in embedded systems.

5.8 Conclusion

In this chapter we have detailed the construction of the Fastcore VM, a portable optimised interpreter. We have highlighted our reasons for our choice of the CVM as a base JVM, targeted for optimisation. We have explained how the implementation was modified with the assistance of Tiger, our interpreter generator. A number of optimisations that we experimented with were described in some detail. Many of these optimisations were highly successful and were retained. Others optimisations that yielded little or no positive improvements were discarded. Finally the results of our new interpreter were presented over a suite of representative benchmarks. The results of Fastcore against other interpreters and JIT compilers has justified our belief that it is possible to construct an optimised portable Java interpreter that can compete with non-portable hand-tuned assembly langauge interpreters such as Hotspot, and on certain benchmarks outperform certain JIT compilers such as the Kaffe JIT on selected benchmarks.

While the optimisations presented in this chapter have yielded some impressive results, they do not represent the full extent of optimisations that can be applied. In subsequent chapters, we present more elaborate and powerful optimisations that improve the performance of Fastcore by a considerable amount.

Chapter 6

Static Instruction Enhancement

6.1 Introduction

While the JVM instruction set is well-defined in terms of the instructions available and their behaviour, there are no restrictions on the actual instructions that may be implemented by a Java interpreter loop. Once bytecode has been read in, it can be modified, have instructions replaced or removed, or even moved around. The important thing is only that the semantics of the original bytecode are preserved.

The JVM instruction set itself comprises of 203 instructions. Even if one is limited to running one byte opcodes, this leaves another possible 53 instructions that could be added to the bytecode at runtime by a virtual machine designer. Typically the type of instructions that might added are quick instructions (Section 6.4.4) which are optimised versions of certain *quickable* bytecode instructions such as getstatic and putstatic.

In direct-threaded code we do not have the same limitation of 53 extra instructions. Instead we can have as many extra instructions as will fit in memory. The 32-bit address of each instruction is now stored in the instruction stream rather than a single byte instruction index. This gives us considerable scope for static instruction set enhancements.

In the rest of this chapter[1] we present various instruction set enhancements that are made to the interpreter core at compile-time. All of these enhancements are the

[1]Some of this work in this chapter has been published in SCOPES 2003 [CGEN03]. A technical report detailing some our more recent work in the area is also available [CGE05a]. These papers describe optimised JVMs constructed and tested by the author of this thesis.

result of careful profiling of Java bytecode. For each class of enhancement presented, we discuss the manner of profiling in some detail as it is critical to the success of these enhancements.

6.2 Profiling Methods

All the instruction enhancements described in this chapter are based on information gained during a profiling phase, running the Fastcore interpreter on various programs. During this phase the interpreter gives information about the bytecodes executed during execution. This information can be used to select the best instructions to add to the JVM in order to improve performance.

Since the method of profiling ultimately determines the selection of instructions to be added to the JVM, we describe the four possible methods here. For the purposes of the explanation, assume the JVM to be enhanced will be run on benchmark X. X will typically be an element of a set, S of benchmarks, for example SPECjvm98. Let $S - X$ represent all other benchmarks in the set of benchmarks.

1. **Dynamic individual**. The profiling data is obtained by running an unoptimised JVM in profiling mode on benchmark X. The data returned is the instruction and instruction sequences, along with the number of times they were executed.

2. **Static individual**. The profiling data is obtained by running an unoptimised JVM in profiling mode on benchmark X. The data returned is the instruction and instruction sequences, along with the number of positions they were encountered in the bytecode.

3. **Dynamic exclusive**. The profiling data is obtained by running an unoptimised JVM in profiling mode on all benchmarks in $S - X$. The data returned is the instruction and instruction sequences, along with the number of times they were executed.

4. **Static exclusive**. The profiling data is obtained by running an unoptimised JVM in profiling mode on all benchmark $S - X$. The data returned is the instruction and instruction sequences, along with the number of positions they were encountered in the bytecode.

Target Benchmark	Individual Profiling	Exclusive Profiling
jack	jack	mpeg, compress, javac, jess, db, mtrt
mpeg	mpeg	jack, compress, javac, jess, db, mtrt
compress	compress	jack, mpeg, javac, jess, db, mtrt
javac	javac	jack, mpeg, compress, jess, db, mtrt
jess	jess	jack, mpeg, compress, javac, db, mtrt
db	db	jack, mpeg, compress, javac, jess, mtrt
mtrt	mtrt	jack, mpeg, compress, javac, jess, db

Table 6.1: Individual versus Exclusive profiling for the SPECjvm98 suite.

Each of the profiling methods have their benefits. For example, when attempting to determine the maximum benefit of an instruction enhancement technique, one should use either the dynamic individual or static individual technique. By using profiling data from either of these techniques, one is tailoring the JVM specifically for that benchmark. For a better indication of how an instruction enhancement might work in general, either of the exclusive techniques would suffice. This is because when we are enhancing a JVM to be run on benchmark X, we will be not be using profiling information garnered from an execution of X, thus avoiding tailoring the JVM specifically for that benchmark.

Table 6.1 illustrates the difference between the individual and exclusive profiling techniques, as applied to the SPECjvm98 suite of benchmarks. Note that when we are testing a new optimisation on the SPECjvm98 suite, it will require seven different JVMs, each targeted towards a particular benchmark in the suite and constructed on the basis of the appropriate profiling data.

6.3 Instruction Specialisation

Many JVM instructions take immediate operands from the instruction stream when executing. Fetching these operands from memory is part of the overhead of interpretation. Specialised instructions are new versions of existing instructions with commonly occurring operands hardwired into them, to reduce operand fetching. Typically, the machine code for a specialised instruction can be much more efficient, not only because it usually eliminates a load, but also because the compiler can optimise the code for

that particular constant.

6.3.1 Implementation

```
GETFIELD_QUICK SP( Object* directObj  -- JavaVal32 result )
               IP(  Uint8 offset -- next );
               {
                   if ( directObject != NULL )
                       result = *(directObject + offset);
                   else
                       NULL_POINTER_EXCEPTION();
               }
```

Figure 6.1: Definition of `GETFIELD_QUICK` VM instruction

Figure 6.1 shows the Tiger definition for `GETFIELD_QUICK`, which is used to fetch the value of a field within an object. This instruction takes an immediate argument from the instruction stream which specifies the offset of the field within the object. `GETFIELD_QUICK` is one of the most frequently executed instructions in our JVM (around 8% of all instructions). The offsets are most commonly one of just a few values, so it may make sense to generate specialised versions for each of these constants.

Figure 6.2 shows the Tiger definition and the generated C code for `GETFIELD_QUICK`, specialised with the immediate operand 0 (a particularly common case). Whereas the generated code would normally load the offset from the instruction stream, in the specialised version it is simply set to the chosen constant. Although this code looks long and complicated, the C compiler will optimise it well. In particular, constant propagation will eliminate the addition of the `offset` entirely in this case.

Interestingly, the JVM instruction set includes quite a number of already specialised instructions. For example, there are four versions of each of the load and store instructions for local variables, for each of the first four local variables. However, we have chosen to convert these specialised instructions into their generic form (de-specialisation), and implement our own generic system for creating specialised instructions. There are three main reasons for this decision.

1. First, by converting to generic versions, we increase the opportunities for using

```
Tiger Code:     +SPEC GETFIELD_QUICK 0
```

```
Generated Code: #define offset 0
                TIGER_SPECIAL_GETFIELD_QUICK_offset_0:
                {
                    Object* directObj;
                    JavaVal32 result;
                    vm_Object$_equals_StackVal32(directObj,SPPTR[-1]);

                    {
                        if ( directObject != NULL )
                            result = *(directObject + offset);
                        else
                            NULL_POINTER_EXCEPTION();
                    }

                    vm_StackVal32_equals_JavaVal32(SPPTR[-1],result);
                    IPPTR=IPPTR+2;

                    goto **IPPTR;
                }
                #undef offset
```

Figure 6.2: Simplified Tiger output for `GETFIELD_QUICK` VM instruction specialised with the immediate operand 0.

superinstructions. For example the sequence `ILOAD_0 IADD` could use the superinstruction `ILOAD-IADD`, whereas it is not practical to have large numbers of specialised superinstructions, such as `ILOAD_0-IADD`.

2. Secondly, the standard specialised instructions in the JVM appear to have been chosen on an *ad hoc* basis, with little attention to how often it appears in real code. For example, the instruction `FSTORE_0` does not appear even once in all the SPECjvm98 benchmarks [Wal99]. We would like to choose the instructions to specialise based on real measurements rather than presumed usefulness.

3. Finally, the immediate operand is not known for many VM instructions until the first time they are executed, so they cannot be specialised in JVM bytecode. For example, the offset for a `GETFIELD` instruction may not be known until the first time it is executed, because it may access another class which might not yet be loaded. Given that field access instructions account for about 16% of executed instructions in the SPECjvm98 benchmarks, this greatly reduces the potential for exploiting specialisation.

Tiger supports specialised instruction in three ways. First, it allows us to automatically generate a version of the interpreter which profiles the value of all immediate operands for each instruction as it executes. Based on this profiling information, we can choose the best combinations of instructions and immediate operands to specialise. Secondly, Tiger generates C source code to implement specialised instructions, from the instruction definitions and the output of the profiler. Finally, Tiger also generates C routines to automatically replace VM instructions and their operands with specialised versions. Thus, almost the entire process of creating specialised instructions is automated.

It is worth mentioning here that this process of introducing specialised instructions into the instruction stream does not affect the list of instructions (Section 5.4.1). This list of instructions is used for superinstruction parsing, contains all instruction indices after de-specialisation and before specialisation. This enables superinstruction parsing to be applied in as many places as possible in the bytecode without specialised instructions interfering in the parsing process.

6.3.2 Specialised Instruction Selection

An important question is how specialised instructions will be chosen. If we want to customise the interpreter for a particular program, we just choose the most commonly executed combinations in a test run of that program. If, on the other hand, the program is not available, then we would like to choose a representative set from profiles of several other programs.

We evaluated the instruction specialisation optimisation using our interpreter system and the SPECjvm98 benchmarks. We selected specialised instructions using three strategies:

1. Counting the dynamically most frequently executed combinations for this particular program (*dynamic individual*)

2. Counting the dynamically most frequently executed combinations in all of the SPECjvm98 programs except this program (*dynamic exclusive*)

3. Counting the most frequent combinations appearing statically in the code of all other programs (*static exclusive*).

Of the three approaches, the latter two are the most realistic. If one was to extend the JVM instruction set by adding specialisations, the most logical approach would be to profile a massive set of Java programs to ensure that the new specialisations occur frequently in as many Java programs as possible. Clearly, this is not feasible for our experiments. The next best solution was to take our working set of benchmarks and profile each of them for potential specialisations.

In contrast, the first approach is doing precisely that; optimising the JVM for the benchmark in question. While this is not a good measure of how specialisations might work in practice, it gives us an upper bound on the performance improvements that specialised instructions would give for that particular benchmark.

Figure 6.3 shows the ten most recommended specialisations for the *db* benchmark based on approach (3). The specialisations are listed in order of decreasing importance, with the most important (i.e. frequently occurring) listed at the top. Note how the standard Java specialised instructions load_0 and load_1[2] are re-introduced first. The

[2]This `load` instruction is a generic version of the Java instructions `aload`, `iload` and `fload`.

first non-standard specialised instruction is a version of vinvokevirtual_quick whose second parameter is being specialised to 2u (the 'u' represents an unsigned literal). The first parameter for this instruction remains unspecialised (denoted by a '?').

```
+SPEC load 0u;
+SPEC load 1u;
+SPEC vinvokevirtual_quick ? 2u;
+SPEC load 2u;
+SPEC vinvokevirtual_quick ? 1u;
+SPEC ldc_quick 0u;
+SPEC load 3u;
+SPEC vinvokevirtual_quick ? 3u;
+SPEC ldc_quick 1u;
+SPEC vinvokevirtual_quick 16u ?;
```

Figure 6.3: Recommended Specialisations for *db* Based on Static Exclusive Profiling

6.3.3 Evaluation

Surprisingly, adding small numbers of specialised instructions to our interpreter actually makes it slower. We used the Pentium 4's hardware performance counters to investigate this. As expected, we found that specialised VM instructions reduce the number of native machine instructions needed to execute the interpreter. Normally, we would expect a corresponding reduction in execution time. However, we also found that specialised instructions also impact on indirect branch prediction rates, which has a much large effect on running time.

Figures 6.4 and 6.5 show the speedups gained by adding specialisations to the interpreter based on *static exclusive* and *dynamic individual* profiling respectively. The number of instructions added at compile-time are plotted on the x-axis. The actual choices of 8, 16, 32, 64, 128, 256 and 512 extra instructions have been chosen to give an indication of how performance changes as more instructions are added to the VM. The y-axis shows the speedup compared to the interpreter without specialised instructions. Both diagrams show signs of added specialisations causing a decrease in interpreter performance. As expected, the *static exclusive* profiling method gave rise to the lesser improvement. Even with 512 specialisations, the JVM is slower than the unspecialised

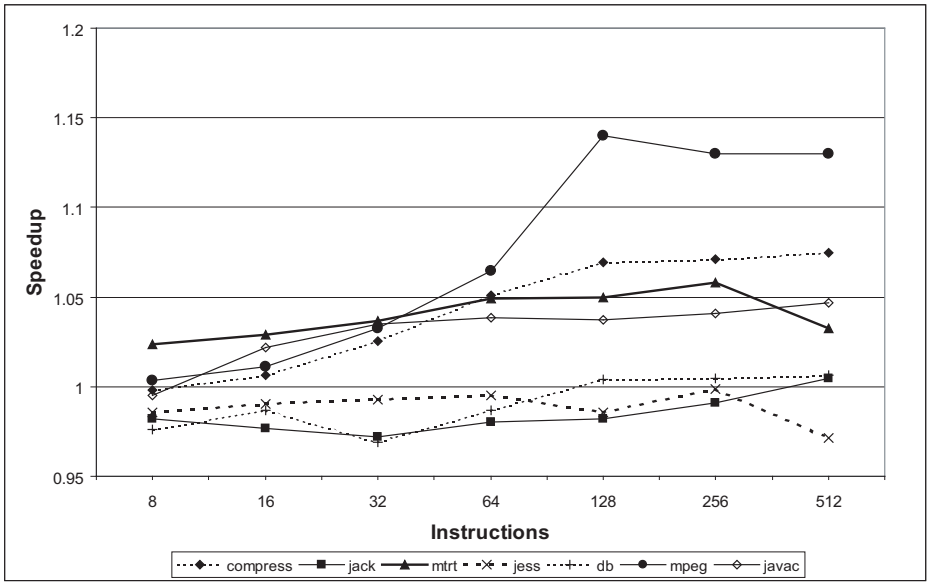


Figure 6.4: Speedup from adding different numbers of specialised instructions chosen based on static frequency in other programs (*static exclusive* profiling).

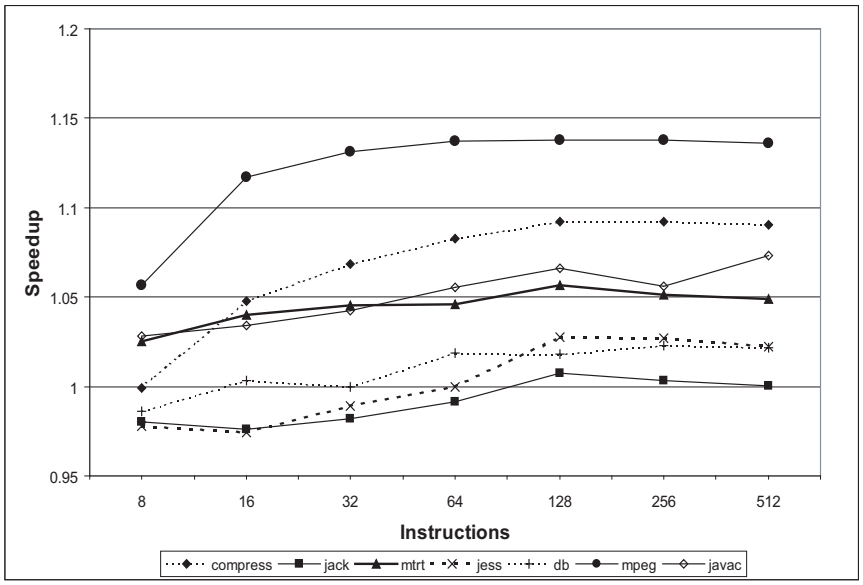


Figure 6.5: Speedup from adding different numbers of specialised instructions chosen specifically for a program (*dynamic individual* profiling).

JVM on the `jess` benchmark. The *dynamic individual* profiling approach does not give substantially better results. This latter profiling method, gives an approximation of the maximum benefit to be obtained from this optimisation.

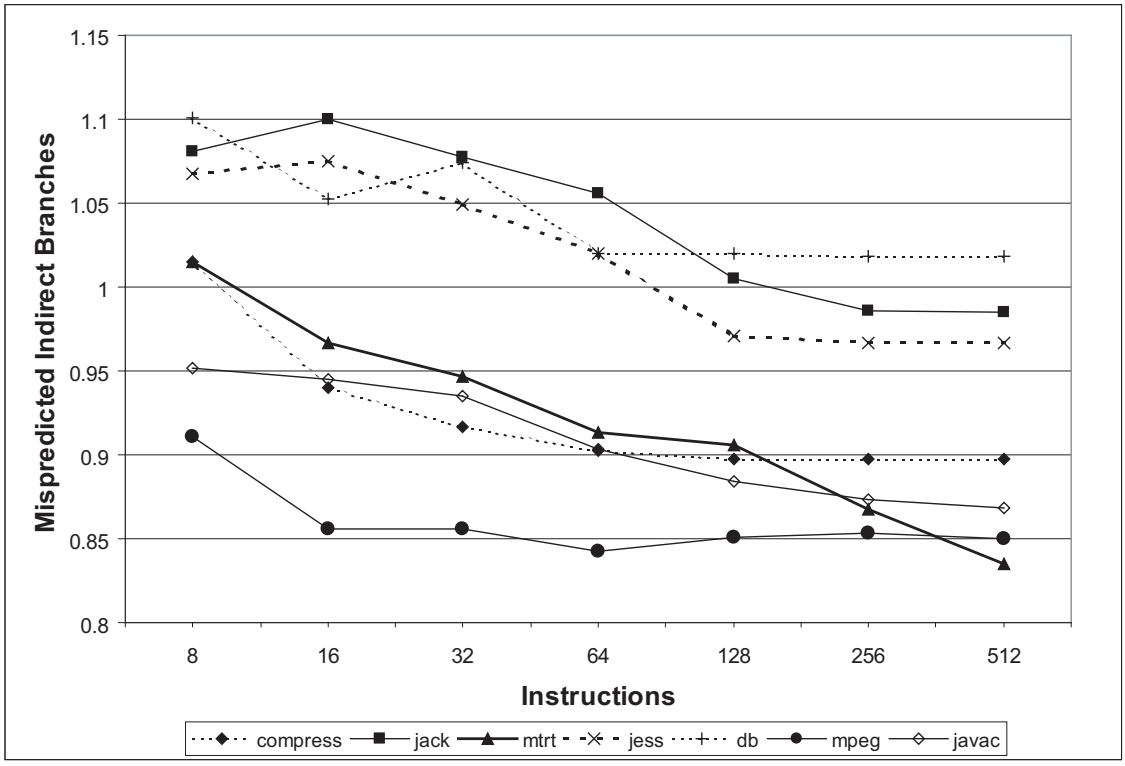


Figure 6.6: Percentage change in indirect branch mispredictions from using specialised instructions chosen specifically for a program (*dynamic individual* profiling).

The small overall improvement in performance was counter-intuitive and further investigation was required. This investigation took the form of measuring the change in branch mispredictions as specialisations were added to the VM. Figure 6.6 shows the ratio of the number of indirect branch mispredictions for various configurations compared to the interpreter without specialised instructions. Small numbers of specialised instructions result in more mispredicted indirect branches. Given the high cost of branch mispredictions (around 20 cycles on the P4) and that dispatching each VM instruction involves executing an indirect branch, a reduction in misprediction accuracy has a much larger effect on execution time than a small reduction in executed instructions.

This misprediction data highlighted the source of the problem. Unlike original JVM bytecode, our base interpreter uses no specialised instructions, and it does not have separate versions of VM instructions for different types, except where they require different code. We made a deliberate decision to minimise the number of VM instructions to facilitate superinstructions. The result is that JVM instructions such as `ALOAD`, `ILOAD` and `FLOAD` are all implemented with the same code. Given that local loads account for around 35% of all executed instructions [Wal99], this code is an extremely frequent target for the indirect branches that implement VM instruction dispatch. When we introduce specialised versions of this local load, we no longer have a single common target, and indirect branch prediction accuracies fall.

However, another effect is also in play. There is a separate indirect branch at the end of the code to implement each VM instruction. When we specialise an instruction, we introduce a new implementation, with its own indirect branch. It is important to recall that current processors use a branch target buffer (BTB) to predict indirect branches, which simply predicts that the target address will be the same as on the previous execution of the same branch. By having a separate indirect branch (and thus a separate BTB entry) for each specialised local load instruction, we capture some context about the program. For example, there may be several `LOAD` instructions in a method, but only `LOAD_6` is followed by `IADD`. As the number of specialised instructions rises, the benefit of more separate indirect branches outweighs the cost of a larger number of targets, and the net effect is positive. We also measured an increase in instruction cache (trace cache) misses, but the effect was much lower than that on indirect branch prediction.

In summary, instruction specialisation is used by many VM interpreters to reduce the overhead of inline immediate operands. However, the main performance impact of specialised instructions is their effect on indirect branch prediction accuracy, rather than reduced operand fetches. Thus, the overall effect is somewhat unpredictable.

6.4 Superinstructions

A superinstruction is a new virtual machine instruction that consists of a sequence of several existing VM instructions. There are a number of benefits associated with this. One is that superinstructions reduce the number of VM instruction dispatches required

to perform a certain sequence of instructions. This is important because instruction dispatch has been shown to be a particular bottleneck in interpreters [EG03b].

Another benefit superinstructions provide is the opportunity to optimise the interpreter source code. For example, it is common that the result written to the stack by one instruction will be read from the stack by the following one. When generating C source code to implement superinstructions, Tiger eliminates the stack read and writes, and instead keeps the value in a local variable between the two component instructions. A third benefit associated with superinstructions is that combining the source code for instructions together exposes a larger "window" of code to the C compiler, which allows greater opportunities for optimisation.

6.4.1 Initial Experiments

There are two main ways that superinstructions are used. The first is to generate an interpreter that is optimised for a particular program. In these initial experiments we will select superinstructions specific to that program. This will, in turn, yield a rough estimate of where the upper bound for performance improvements might lie before we turn to more realistic and universal superinstruction selection strategies in the next section.

Selecting the optimal set of superinstructions for a given program is NP-hard [Pro95]. We experimented with a number of heuristics, such as finding the most frequently executed (sub)sequences of VM instructions, in a scheme similar to Proebsting's [Pro95]. Eventually we found that by far the best scheme is to simply select the n most frequently executed basic blocks in the program to be superinstructions.

The set of superinstructions to be used were based on this particular scheme, operating on profiling data for each benchmark separately. Thus we used the *dynamic individual* method of profiling (Section 6.2). Using this profiling data, we built JVMs incorporating superinstructions built from the 8, 16, 32, 64, 128, 256 and 512 most commonly executed basic blocks. This approach was tested using the SPECjvm98 benchmarks. Results are presented in Figure 6.7. These results are presented in terms of speedup over the same implementation of the JVM and benchmark with 0 superinstructions.

Although this individual tailoring of JVMs for each benchmark seems a little un-

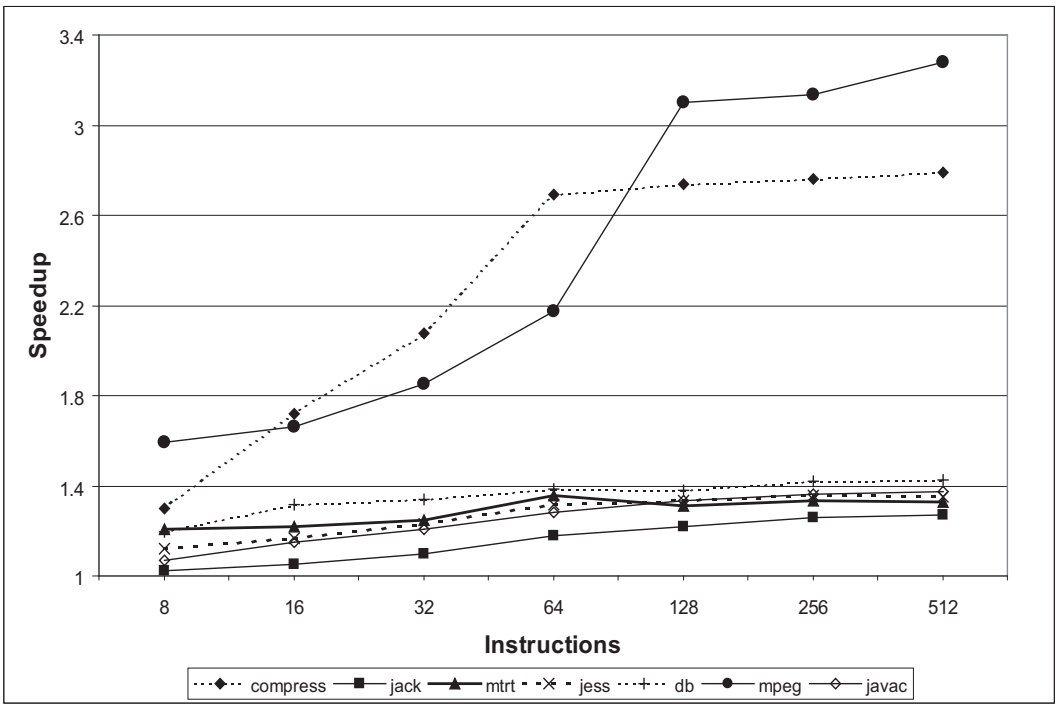


Figure 6.7: Adding individually tailored superinstructions to the interpreter (*dynamic individual* profiling).

realistic it has the major benefit that it gives something of a yardstick against which to measure performance of other, more realistic, schemes. Additionally, it gives some measure of the degree to which a JVM can be optimised to a particular task, a proposal made by Venugopal et al. [VMK02].

Looking at Figure 6.7, the speedups achieved using varying numbers of superinstructions are apparent. The figures are quite impressive for some benchmarks, in particular *compress* and *mpeg*, with speedups of 2.8 and 3.3 respectively at 512 superinstructions. Other benchmarks do not benefit so well from this approach but nonetheless the minimum speedup is 1.27 (*jack*) at 512 superinstructions which is quite reasonable. It is worthwhile noting that, even with only 32 superinstructions, the minimum speedup across all benchmarks is 1.1.

6.4.2 Which Sequences?

The main determinant of the usefulness of superinstructions is whether the sequences we choose to make into superinstructions account for a large proportion of the running time of the programs that run on the interpreter. The set of superinstructions must be chosen when the interpreter is constructed, most likely at a time when one doesn't know which programs will be run on the interpreter. Thus, one must somehow guess which superinstructions are likely to be useful for a set of programs that one has never seen.

The most common way to make guesses at the behaviour of unseen programs is to measure the behaviour of a set of standard benchmarks programs, and hope that these benchmarks resemble the real programs. A question remains, however, as to how the benchmarks should be measured to identify useful superinstructions.

We tested several criteria for selecting superinstructions. We tested measuring the *static* number of times that each sequence appears in the code as well as its *dynamic* execution frequency. In order to avoid one large program dominating the others, we *normalised* the frequencies to percentages of total static/dynamic instructions in the program. Originally, we felt that longer sequences were more desirable, because they eliminate more dispatches, so we tried multiplying the frequencies by the length of the superinstruction minus one (*lmul*). It also occurred to us that shorter superinstructions might be easier to reuse, so we also tried dividing by the number of dispatches removed

Scheme	8	16	32	64	128	256	512
static	26.3	27.7	30.7	33.0	38.7	41.6	44.2
static norm.	26.3	27.3	30.7	33.6	38.8	40.8	44.0
static ldiv	23.7	28.1	30.3	34.5	39.6	44.5	48.3
static ldiv norm.	23.7	27.8	29.8	34.4	38.7	44.2	47.4
static lmul	16.7	17.9	19.0	19.1	19.7	21.0	21.3
static lmul norm.	19.3	20.4	21.3	23.5	24.6	25.1	25.8
dynamic	24.1	26.0	30.2	31.8	33.2	34.9	36.5
dynamic norm.	23.8	28.8	32.4	35.4	37.7	42.3	43.4
dynamic ldiv	23.8	26.4	29.7	33.6	37.4	41.2	44.1
dyn. ldiv norm.	23.8	26.7	30.9	35.7	40.8	44.4	47.2
dyn. lmul	1.3	1.3	1.3	2.5	2.6	2.9	3.0
dyn. lmul norm.	15.5	15.9	17.3	18.5	18.6	19.8	20.4

Table 6.2: Comparison of superinstruction selection strategies.

by the superinstruction (*ldiv*). Table 6.2 shows the average reduction in VM instruction dispatches across all benchmarks using different combinations of the selection strategies and varying numbers of superinstructions. A number of trends are clear. Dynamic frequency performs worse than static, because it is biased very strongly in favor of the inner loops of the programs.

The argument in favour of static selection is also made in Gregg and Waldron [GW02] where a wide range of strategies were tested for choosing superinstructions for Forth programs. They found, perhaps surprisingly, that the best strategy was to simply choose those sequences that appear most frequently in the static code. This was the next strategy we chose. More specifically, we chose the *static exclusive* profiling method (Section 6.2) to generate a list of most commonly occurring basic blocks. These basic blocks were used to create superinstructions to be added to the optimised JVMs.

Results are presented in Figure 6.8 for this strategy. The results are graphed similarly to before, with results presented for each benchmark with a varying number of superinstructions. The speedups obtained with this approach were much more conservative. At 512 superinstructions, the maximum speedup was 1.75 (*mpeg*) and the minimum 1.07 (*jack*). At 32 superinstructions the maximum speedup was 1.36 (*mpeg*) and the minimum 1.05 (*jack*).

Analysis of the superinstructions selected using the strategy above yielded some

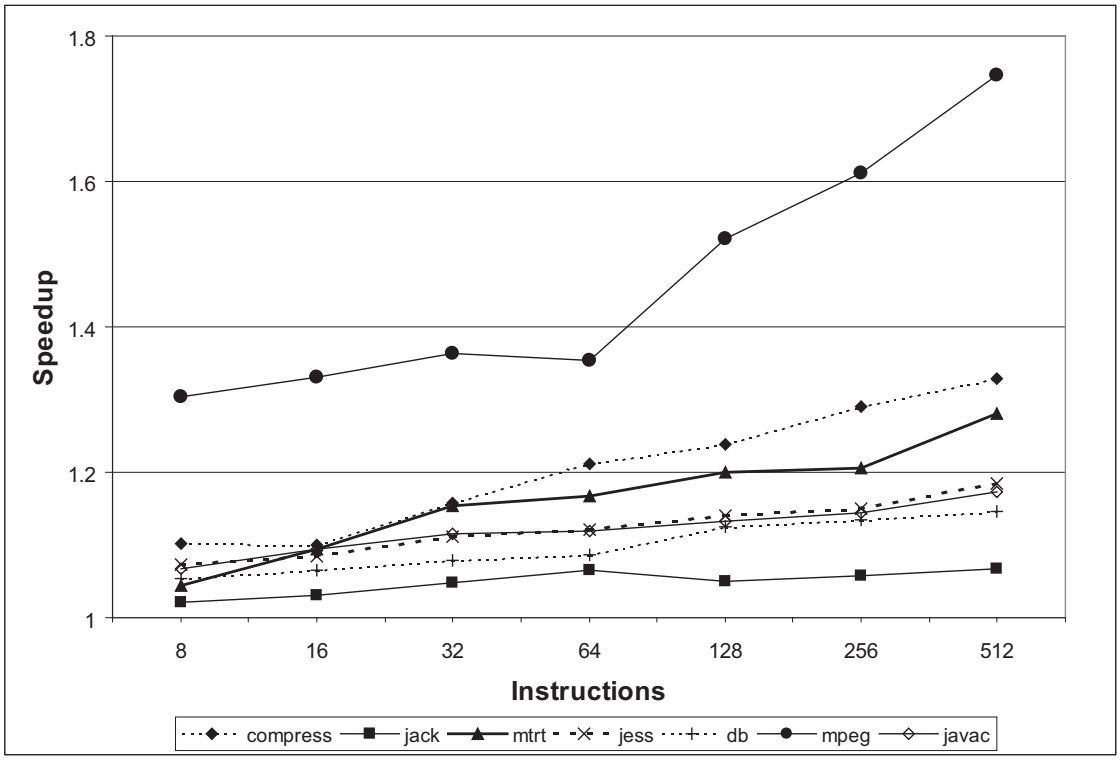


Figure 6.8: Adding statically selected superinstructions to the the interpreter (*static exclusive* profiling).

interesting results. It appeared that some long sequences from a limited number of benchmarks were dominating the statistics. In an attempt to reduce this effect, we decided to bias the statistics in favor of shorter, and therefore more commonly occurring (across benchmarks), sequences. In order to do this, we used precisely the same superinstruction selection strategy as before, but this time each superinstruction's weight (previously its static frequency) was divided by its length-1. This biases the selection strategy heavily in favor of shorter sequences.

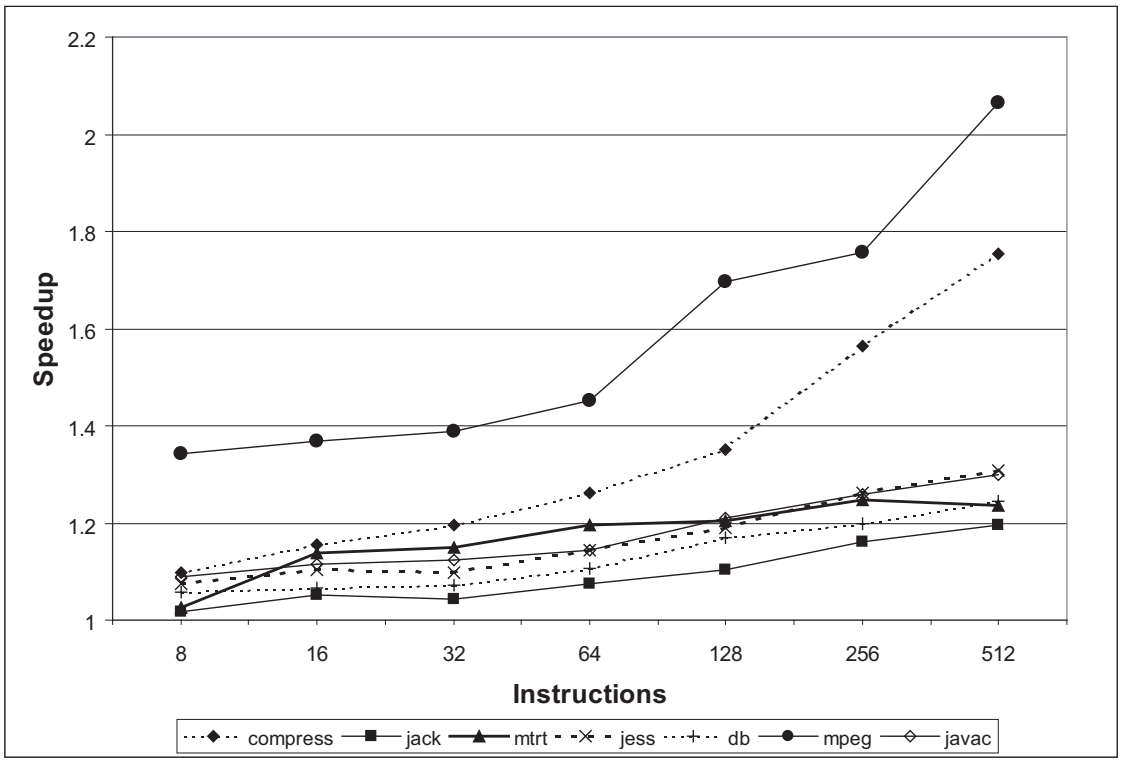


Figure 6.9: Adding statically selected *short* superinstructions to the the interpreter (*static exclusive* profiling).

Results were generated exactly as before, but this time using the modified weightings to decide which superinstructions to include. The results are presented in Figure 6.9. The speedups were considerably better with this minor modification. At 512 superinstructions, the maximum speedup was 2.06 (*mpeg*) and the minimum 1.19 (*jack*). At 32 superinstructions the maximum speedup was 1.39 (*mpeg*) and the minimum 1.04 (*jack*). These results show two interesting points. Firstly, the superinstruction

selection scheme is critical. Even small changes in the selection algorithm can have dramatic effects. Secondly, there are two opposing goals in that we would like choose long superinstructions (to eliminate as many dispatches as possible) but shorter superinstructions can be applied at more points in the code. More sophisticated selection algorithms will need to be examined to throw more light on this aspect of superinstructions.

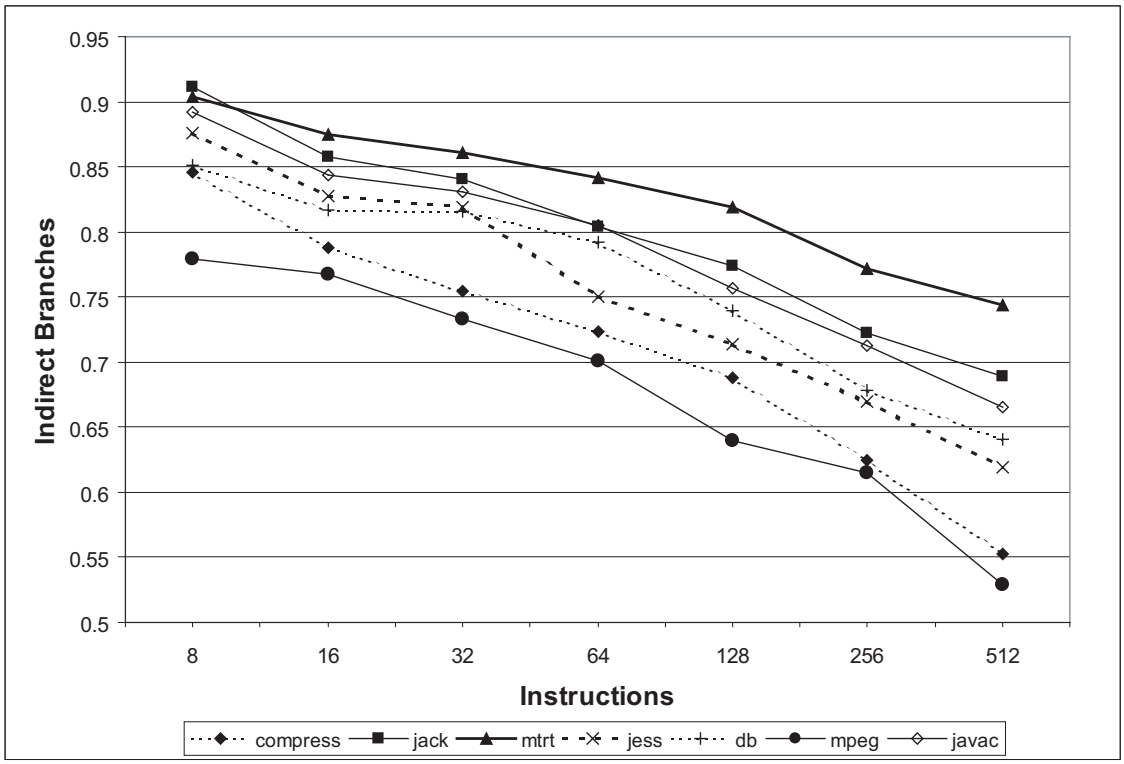


Figure 6.10: Indirect branch reduction due to statically selected short superinstructions (*static exclusive* profiling).

Superinstructions, by virtue of eliminating dispatches, reduce the number of indirect branches and consequently the number of indirect branch mispredictions. This is illustrated in Figure 6.10 and Figure 6.11 respectively. These represent indirect branch measurements and mispredicted indirect branch measurements for the same benchmarks and selection strategy presented in Figure 6.9. Normally one would expect that the misprediction of indirect branches as a proportion of indirect branches stays more or less constant as superinstructions are added to the JVM. If this were the case, both

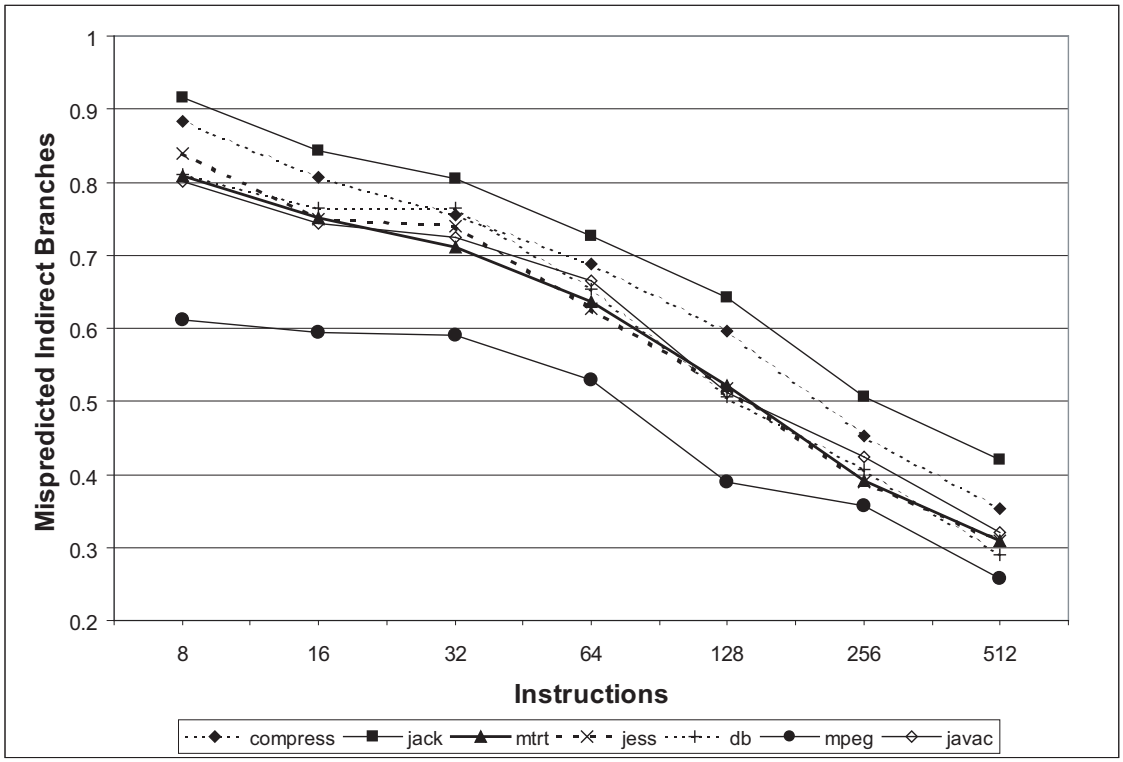


Figure 6.11: Mispredicted indirect branch reduction due to statically selected *short* superinstructions (*static exclusive* profiling).

the indirect branch counts and mispredicted indirect branch counts would be decreasing at the same rate as superinstructions are added. However Figures 6.10 and 6.11 illustrate that the number of indirect branch mispredictions decreases more sharply than the number of indirect branches. From an average misprediction rate across all benchmarks with 0 superinstructions of 45%, the rate drops substantially, down to a little over 23% at 512 superinstructions.

The explanation for this is twofold. The addition of each superinstruction adds an extra entry to the Branch Target Buffer (subject to BTB size). Also, by the time a dispatch occurs at the end of a superinstruction, we are guaranteed to have executed a certain sequence of component instructions. This has the effect of adding context to the dispatch at the end of the superinstruction in question, and makes branch prediction at that point more accurate.

One complication in a Java interpreter is that the JVM comes with a large library of classes that are used internally by the JVM and by running programs. Approximately 33% of the executed bytecode instructions in the SPECjvm98 benchmark suite [SPE98] are in library rather than program methods [Wal99]. This library code is available at the time the interpreter is built, so there is potential for choosing superinstructions specifically for commonly used library code.

As with other optimisations, we use the interpreter generator Tiger to generate superinstructions using profiling information. `Tiger` allows for some extra functionality over vmgen including support for superinstructions across basic blocks.

One important feature Tiger allows for is the generation of superinstructions from profiling information. For example, we might find that the VM instructions `ILOAD` and `IADD` occur very frequently in sequence in Java programs. Given their frequency, it may be worthwhile to create a superinstruction `ILOAD-IADD` which behaves exactly like the original sequence of VM instructions but may be more efficient.

Figure 6.12 shows the instruction definition for the JVM instruction `ILOAD` (load integer local variable). Note that we need to update the instruction pointer by two positions, since the VM instruction consists of the `ILOAD` opcode followed by an immediate operand containing the number of the local variable to load onto the stack.

By adding `ILOAD-IADD` to the list of superinstructions for our interpreter, Tiger will produce the source code in Figure 6.13, which is generated automatically from the instruction definitions of `ILOAD` and `IADD`.

```
ILOAD SP( -- Int32 result )
      IP( Uint8 index );
{
    result = locals[index];
}
```

Figure 6.12: Definition of `ILOAD` VM instruction

There are a number of notable features about this code. First, all used stack items are loaded from memory into local variables at the start of the code. The different VM instructions within the superinstruction communicate by reading from and assigning to these local variables.

Presuming that the C compiler is able to allocate these local variables to registers, this will greatly reduce the amount of memory traffic from accessing the VM stack. `IADD` alone requires two loads and one store to access the stack, and `ILOAD` requires one store. In contrast, the superinstruction `ILOAD-IADD` requires only one load and one store access to the stack to perform the same work. Thus stack memory traffic is reduced by 50%.

Another notable feature of the code in Figure 6.13 is that there is no stack pointer update. `ILOAD` increases the size of the stack by one, and `IADD` reduces its size by one. Vmgen detects that the two stack pointer updates are redundant, and eliminates them. In addition, there is only one instruction pointer update.

6.4.3 Parsing

The use of superinstructions is in many respects the same problem as dictionary-based text compression [BCW90]. Dictionary-based compression attempts to find common sequences of symbols in the text, and replaces them with references to a single copy of the sequence. Thus, when designing a superinstruction system, we can draw on a large body of theory and experience on text compression.

Parsing is the process of modifying the original sequence of instructions by replacing some subsequences with superinstructions. The simplest strategy is known as *greedy parsing*, where at each VM instruction we search for the longest superinstruction that will match the code from that point.

For example, consider the basic block in Figure 6.14. Assume that we have two

```
ILOAD_IADD: /* start label */
{
Int32 sp0;  /* synthetic names */
Int32 sp1;
Int32 ip1;  /* synthetic name for item in VM instruction stream */
ip1 = *(ip+1); /* fetch immediate value */
sp0 = *(sp);
{ /* ILOAD */
  Int32 iIndex; /* declare stack item */
  Int32 iResult;
  /* fetch stack item to local variable */
  iIndex = ip1;
  {               /* user provided C code */
    iResult = locals[iIndex];
  }
  sp1 = iResult;  /* store stack result */
}
{ /* IADD */
  Int32 iValue1;    /* declare stack items */
  Int32 iValue2;
  Int32 iResult;
  iValue1 = sp1;  /* fetch stack items to */
  iValue2 = sp0;  /* ...local variables */
  {               /* user provided C code */
    iResult = iValue1 + iValue2;
  }
  sp0 = iResult;  /* store stack result */
}
*(sp) = sp0;
ip += 3;          /* update VM ip */
}
NEXT;             /* indirect goto */
```

Figure 6.13: Simplified Tiger output for `ILOAD-IADD` superinstruction

```
ILOAD 4   ; load local 4
ILOAD 5   ; load local 5
IADD      ; integer add
ISTORE 6  ; store TOS to local 6
ILOAD 6   ; load local 6
IFEQ 7    ; branch by 7 if TOS == 0
```

Figure 6.14: Example basic block

superinstructions available: `ILOAD-ILOAD` and `ILOAD-IADD-ISTORE`. Following a greedy strategy, we would find the longest sequence that matches a superinstruction from the start of the basic block. Thus, we would replace the first two instructions with the superinstruction `ILOAD-ILOAD`, and reduce the number of dispatches needed to execute this code by one. The main advantage of greedy parsing is that it is very fast — an important factor in an optimisation that we apply to a Java method at run time, the first time that it is invoked. Greedy parsing is also simple to implement and requires little memory.

The weakness of greedy parsing becomes apparent when we consider whether a better parse of the code in figure 6.14 is possible. Clearly, it would be better to replace the second, third and fourth instructions with the superinstruction `ILOAD-IADD-ISTORE`. This would reduce the number of dispatches by two.

To be guaranteed to find the best possible parse, an optimal parsing algorithm must be used. Fortunately, optimal parsing can be solved using dynamic programming [BCW90], so efficient algorithms are available. Our interpreter currently allows for either greedy or optimal parsing to be selected at compile time. Tiger generates parsing tables that remove this requirement, so we are free to add superinstructions to a JVM without ensuring all subsequences are present. This will allow us to exploit more advanced superinstruction selection strategies.

All results in this chapter are presented using the optimal parsing algorithm except for those presented in Figure 6.15 where we present a comparison of greedy parsing versus optimal parsing for the same selection strategy as used in Figure 6.9. This comparison shows that there is not a huge difference between optimal parsing and greedy parsing in terms of performance. Indeed sometimes greedy gives a better speedup. The extra computational overhead required for an optimal parse is the most likely reason

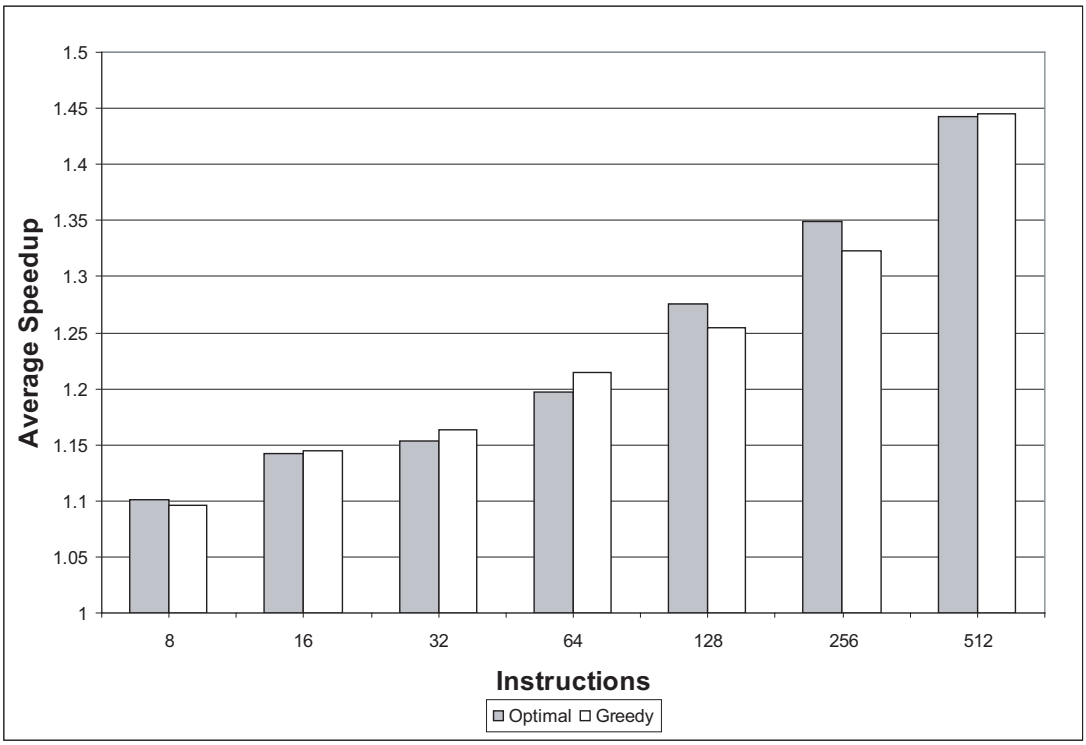


Figure 6.15: Comparison of optimal versus greedy parsing strategies for statically selected superinstructions (*static exclusive* profiling).

for this, where it occurs. It can be quite possible that, with superinstruction selection strategies other than that used for the comparison in Figure 6.15, that the difference between an optimal parsing process and a greedy one would be more noticeable.

6.4.4 Quickable Instructions

Several Java bytecode instructions must perform various class initialisations the first time that they are executed. On subsequent executions no initialisations are necessary. A common way to implement this functionality is with "quickable" instructions. The first time a given instruction of this type is executed, it performs the necessary initialisations, and then replaces itself in the instruction stream with a corresponding quick instruction, which does not do these initialisations. On subsequent executions of this code, the quick instruction is executed.

Quick instructions are vital to the performance of most Java interpreters, since the check for class initialisation is expensive, and because they are among the most commonly executed instructions. For example, in the SPECjvm98 benchmarks `GETFIELD` and `PUTFIELD` account for about one sixth of all executed instructions, and run very slowly unless converted to quick versions [Wal99]. Eller [Ell99] found that adding quick instructions to the Kaffe interpreter could speed it up by almost a factor of three.

A problem with quickable instructions is that they make it difficult to replace sequences of instructions with superinstructions. No instruction that will be replaced with another instruction at run time can be placed in a superinstruction, since that would involve replacing the entire superinstruction. Furthermore, some instructions, such as `LDC` (load constant from constant pool) and `INVOKEVIRTUAL` become different quick instructions depending on the value of their inline arguments, or the type of class or method they belong to.

An additional complication when dealing with quickable instructions is race conditions. The Java interpreter supports multiple threads of execution within a single program. Therefore, during quickening it is quite possible for two threads to almost simultaneously access a non-quick instruction triggering a potential race condition. Such race conditions are avoided in the current implementation of the JVM by using mutually exclusive locks, but adding support to allow quickened instructions to become part of a superinstruction after translation could lead to race conditions.

Our current implementation allows for "quick" instructions to be components in superinstructions by utilising a simple approach. Each time an opcode is quickened in a method, the method is re-parsed in an attempt to incorporate that newly created quick instruction into a superinstruction. There is a computational cost associated with the approach but we have not found it to be significant. Nonetheless, if one wanted to reduce the parsing overhead, a possibility is to re-parse only the basic block in which the instruction has been quickened. The difficulty with this approach, however, is that we have the capability of having superinstructions that span basic blocks (see Section 6.4.5 below). Thus we cannot limit a re-parsing to the basic block in which an instruction was quickened if we want to attempt to use these longer superinstructions.

The value of permitting quickened instructions to be part of superinstructions can be seen empirically, by comparing the data in Figures 6.8 and 6.7 with Figures 6.16 and 6.17 respectively. Figures 6.16 and 6.17 emanate from a previous implementation of our interpreter [CGEN03] where quick instructions were not permitted to be part of superinstructions. Apart from the non-quickable aspect, the method of superinstruction selection is almost identical. Figure 6.16 represents the *static exclusive* profiling based selection method, while Figure 6.17 shows the *dynamic individual* profiling based selection method.

Comparing like with like, it can be seen from Figure 6.8 that allowing quickened instructions inside superinstructions allows a substantial improvement over the non-quickable equivalent presented in Figure 6.16. For example the speedup for *mpegaudio* jumps from just under 1.45 without quick superinstructions to over 1.7 with quick superinstructions. Comparing the dynamic-individual tailored methods of superinstruction selection, it can be seen from Figures 6.7 and 6.17 that permitting quickened instructions using this method of superinstruction has an even greater effect. In this case the speedup for *mpegaudio* jumps from just under 1.75 to approximately 3.3 by permitting quickened instructions in superinstructions.

6.4.5 Across Basic Blocks

Superinstructions are normally only applied to instructions within basic blocks. However, with relatively small modifications, it is possible to extend superinstructions across basic block boundaries in two specific situations. First, we consider control

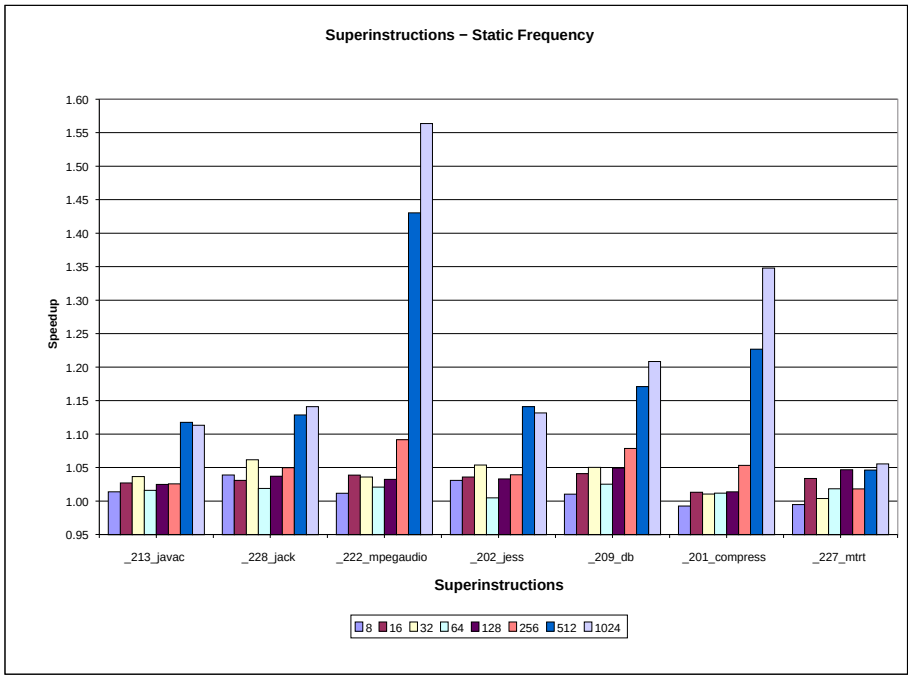


Figure 6.16: Adding statically selected non-quick superinstructions to our interpreter.

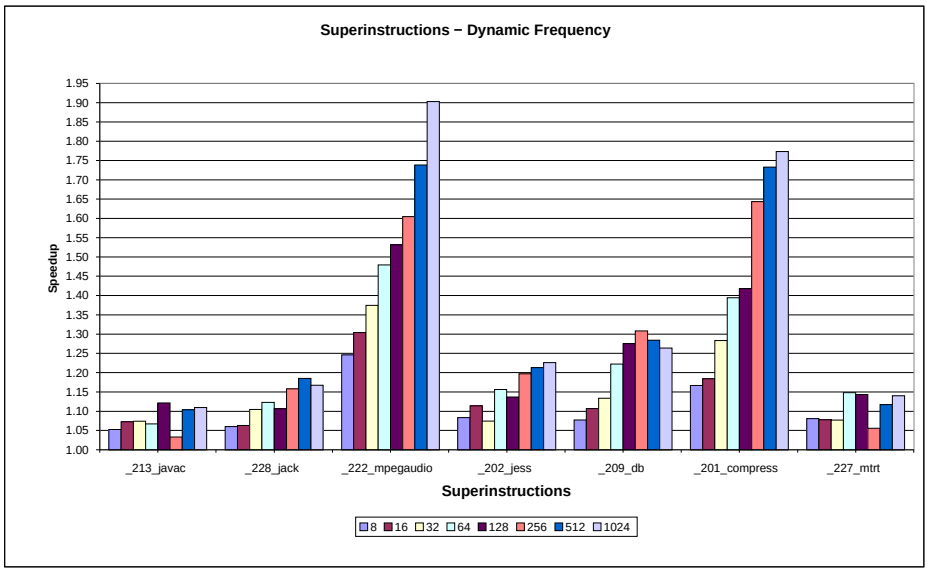


Figure 6.17: Adding individually tailored non-quick superinstructions across basic blocks to our interpreter.

flow joins. A join is a point in the program with incoming control flow from two or more different places. Usually one of those places is simply the preceding basic block, and control falls through to the join without any branching. In these cases, the falling-though code is simply a straight-line sequence of instructions. However, it is not normally safe to allow a superinstruction to be formed across the join, because it would not then be clear where the other incoming control-flow paths should branch to.

```
                ILOAD                       ILOAD
                4                           4
                ILOAD                       ILOAD-IADD
                5                           5
        join:   IADD                join:   IADD
                ISTORE                      ISTORE
                6                           6
```

Figure 6.18: Original bytecode (left) and same bytecode with `ILOAD-IADD` superinstruction (right).

The solution we use is to create superinstructions, but not to remove the gaps that are created by eliminating the original instructions. In fact, we leave the original instructions in these gaps. Figure 6.18 shows an example where we have replaced the sequence `ILOAD`, `IADD` with the superinstruction `ILOAD-IADD`. We actually replace the `ILOAD` instruction with `ILOAD-IADD`, but leave the IADD instruction where it is. When we fall-through from the first basic block to the second, we execute `ILOAD-IADD`, which performs its normal work and then skips over the `IADD` instruction[3]. On the other hand when we branch to the second basic block from elsewhere, we branch to the `IADD` instruction which executes and continues as normal. This scheme allows us to form superinstructions across fall-though joins.

We believe that this scheme is particularly valuable for `while` loops. The standard *javac* code generation strategy appears to be to place the loop test at the end of the loop, and on the first iteration to jump directly to this test. Unfortunately, the result is that there is a control flow join just before the loop test that would normally hinder optimisation. We believe we have successfully overcome this problem.

[3]The `ILOAD-IADD` instruction increments the virtual program counter by 3, thus making it point to the `ISTORE` instruction.

```
ifnull SP(  Object* o1 - )
       IP( +DEFER Int32 skip - +DEFER next);

        if((o1 == 0))
        {
                VMLOAD_skip_;
                SET_IP(pc+skip);
        }
        VMLOADnext;
```

Figure 6.19: Definition of a branch VM instruction

A second opportunity for cross-basic block superinstructions is with the fall-through direction of VM conditional branches. Superinstructions are permitted to extend across branches due to facilities provided by Tiger. Figure 6.19 shows the instruction definition for a branch instruction. Inside the `if` statement the Tiger keyword `SET_IP` is used to specify that a copy of the dispatch code that normally appears at the end of the instruction should be placed here.

The `SET_IP` macro in Tiger is redefined at the beginning of every component instruction in a superinstruction to allow a branch out of the superinstruction at that point, if required. Thus a single superinstruction can be generated that spans multiple untaken branches. Of course if the branches are taken, the `SET_IP` macro will flush items from local variables to the stack and update the stack pointer before a branch out of he superinstruction.

Tiger also redefines a `FLUSH_ALL` macro that carries out a similar task to `SET_IP` (but without the dispatch) before each instruction. This was necessary for certain 'superinstruction-unsafe' instructions such as `ANEWARRAY_QUICK` which may trigger a garbage collection. When a garbage collection occurs, the stack pointer and items on the stack should be flushed from their cached positions in local variables, back to their appropriate position in memory. Previously, we could not allow such 'superinstruction-unsafe' instructions to be components in a superinstruction because the stack pointer had not been up to date and stack items were still in local variables (ready to be written later in the superinstruction). Now, if the `FLUSH_ALL` macro is used before such instructions, all items will be flushed to the operand stack before the instruction (and hence before the garbage collection takes place). Elimination of stack pointer updates

and accesses using local variables can resume for subsequent component instructions in the superinstruction after the 'superinstruction-unsafe' instruction has completed Using this mechanism, a large number of instructions that previously were unsafe to include in superinstructions can now be used.

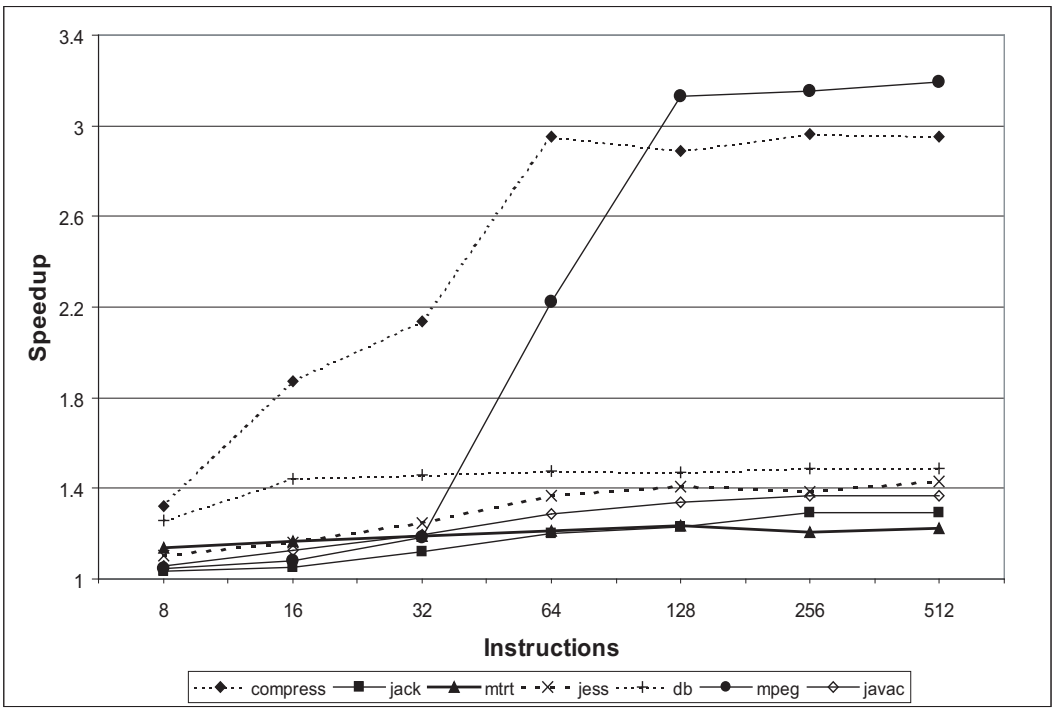


Figure 6.20: Adding individually tailored superinstructions across basic blocks to the interpreter (*dynamic individual* profiling).

Using a similar method to select superinstructions as that in Figure 6.7 we created a set of JVM binaries containing superinstructions tailored to each individual benchmark. The difference is that instead of using the most commonly occurring basic blocks we now use the most commonly occurring regions. The start point for these regions can be any join point and the end point can be any branch or invocation. The results are presented in Figure 6.20. The speedups obtained by using the longer regions instead of shorter basic blocks appear to be reasonable. Most benchmarks experienced a speedup with the exception of *mtrt*. Altogether, 31 of 49 JVM runs registered an improvement over basic block selection (7 benchmarks with 7 runs per benchmark). A superinstruction selection strategy tailored to exploit this new capability of superinstructions across

basic blocks can yield even better results.

6.5 Instruction Replication

Instruction replication aims to reduce indirect branch mispredictions in a threaded code interpreter by creating multiple versions of the code to implement commonly occurring VM instructions. Each separate version is ended by an indirect branch to dispatch the next VM instruction. By varying the version of the implementation code that is used in different parts of the Java program, some context information is captured which may improve branch prediction accuracy.

For example, consider a loop containing the following sequence of instructions: `ILOAD IADD ILOAD ISUB`. If there is only one version of the `ILOAD` code, then the indirect branch at the end of the code to implement `ILOAD` will constantly switch between the two and indirect branch prediction accuracy will be very poor. If, on the other hand, we have two versions of `ILOAD`, then this could could be rewritten as `ILOAD_A IADD ILOAD_B ISUB`. There are separate indirect branches at the end of each replication, both of which are now almost perfectly predictable. Figure 6.21 illustrates this

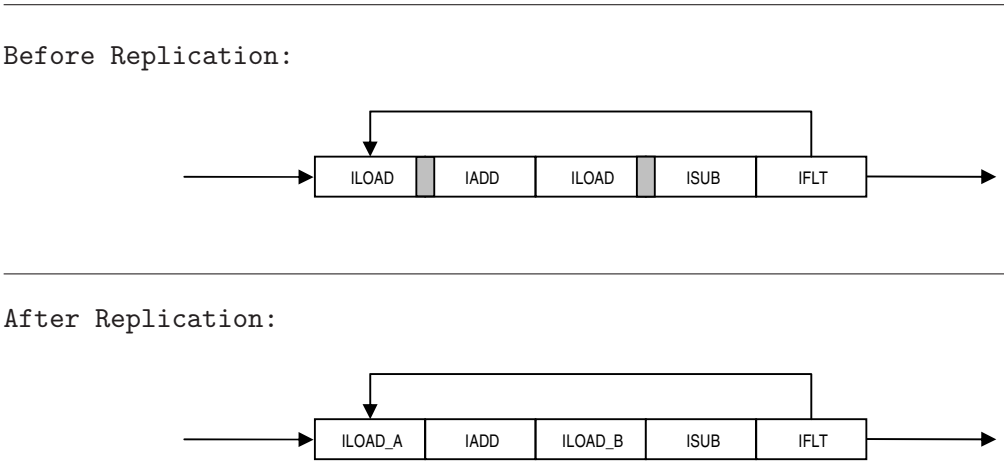


Figure 6.21: Adding static replications to improve branch prediction.

loop before and after `ILOAD` has been replicated. Note the grey transitions before repli-

cation is employed. These are dispatch points that will nearly[4] always cause a branch misprediction. Note how the dispatches are much more predictable after replication. `ILOAD_A` is always followed by `IADD` and `ILOAD_B` is always followed by `ISUB`.

6.5.1 Implementation

Tiger supports instruction replication by:

1. generating a profiling version of the interpreter to collect profiles of instruction frequencies,
2. automatically generating copies of the C source code to implement VM instructions and,
3. generating C source code to rewrite the VM code with replicated instructions.

When several versions of a VM instruction exist, the versions are used in round-robin order when rewriting the VM code. This usually ensures that the same version of a VM instruction is not used more than once in a basic block.

An important question is which VM instructions should be chosen for replication, and how many copies of each should be created. Based on profiles, we computed the frequency of each VM instruction. We added one replication of the most frequent VM instruction, and reduced its frequency by splitting the frequency between each of the copies. We then applied the same process again, until we had chosen the required number of total replications. Note that this process can result in the same instruction being replicated multiple times, because even its split frequency may be higher than that of other instructions.

6.5.2 Evaluation

We evaluated a number of combinations of various parameters. The frequencies of instructions were measured by both their static and their dynamic occurrences. The interpreter can be optimised for a given program, or profiling data from several programs can be used to find generally useful sets of replicas. We found that there was

[4]except the first time the loop is executed (*cold misses*), and also occasionally when BTB entries change due to code executing in other threads.

almost no additional benefit from customising the replicated instructions for a particular program. Static measures of frequency perform a little worse than dynamic measures (see Figures 6.22 and 6.23 for comparison).

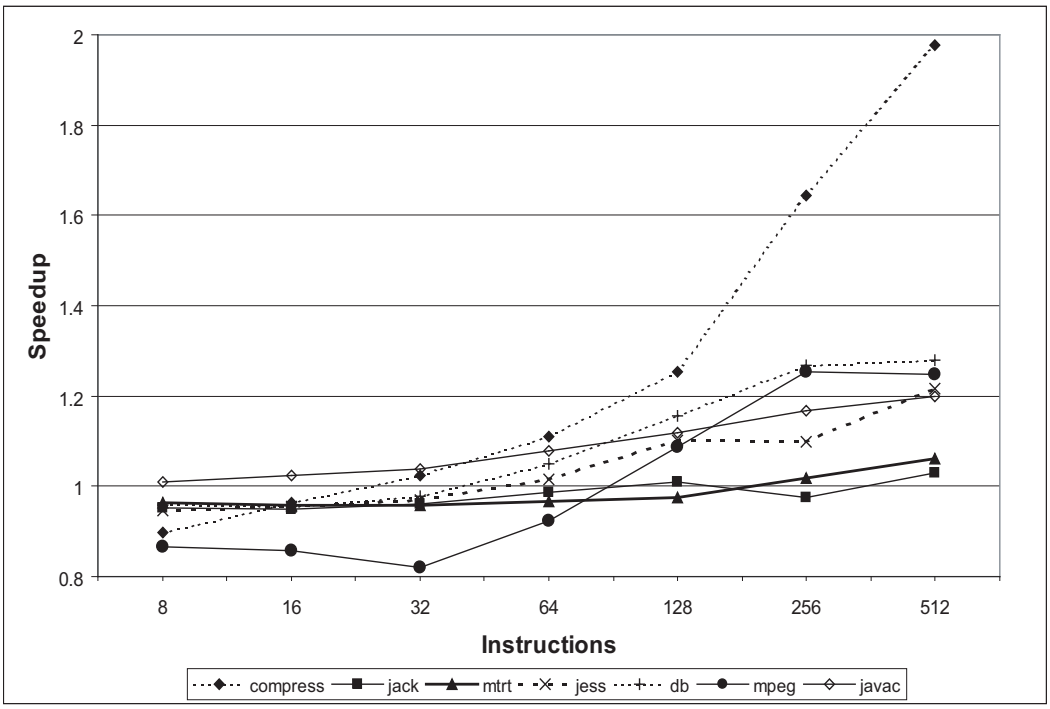


Figure 6.22: Speedup from replicated instructions chosen using dynamic frequency in other programs.

Figure 6.22 shows the effect on running time of using varying numbers of instructions selected based on their dynamic frequency in all SPECjvm98 programs except the one being measured. As with instruction specialisation, adding small numbers of replicated instructions actually makes the interpreter slower. As in Section 6.3, we investigated this using the Pentium 4's hardware instruction counters, and found that adding small numbers of replications increases the branch misprediction rate (see Figure 6.25), because of the very frequent local load instruction being replicated. There was also a significant increase in the number of instruction cache (trace cache) misses. As the number of replications increases, the reduction in indirect branch mispredictions outweighs the other costs, and there is a significant net speedup (almost a factor of two for compress).

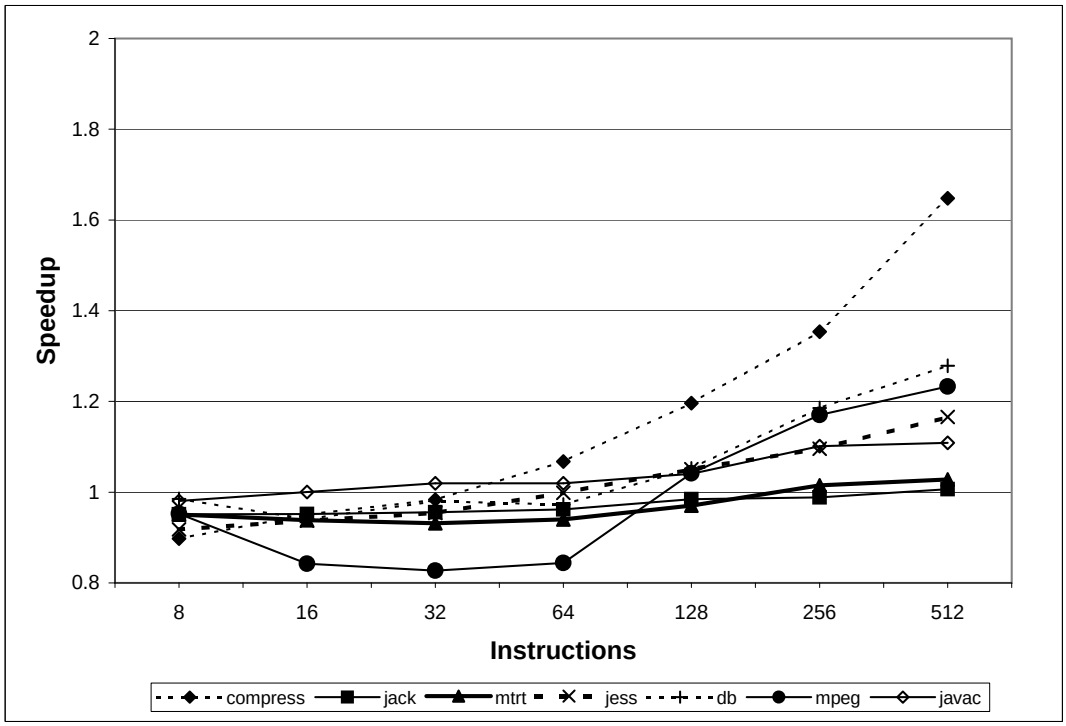


Figure 6.23: Speedup from replicated instructions chosen using static frequency in other programs.

```
+ALIAS fload 1;
+ALIAS fload 1;
+ALIAS fload 1;
+ALIAS ldc_quick 1;
+ALIAS fload 1;
+ALIAS fstore 1;
+ALIAS fload 1;
+ALIAS agetfield_quick 1;
+ALIAS fload 1;
+ALIAS getfield_quick 1;
```

Figure 6.24: Recommended Replications for *db* Based on Dynamic Exclusive Profiling

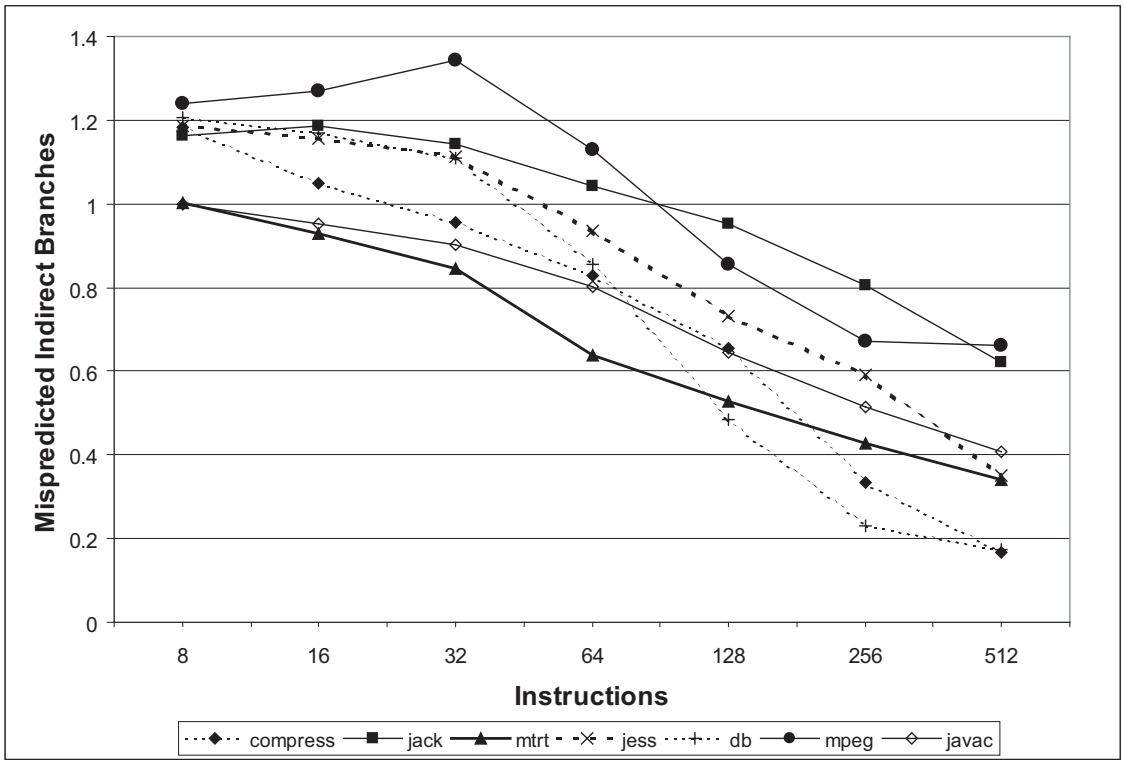


Figure 6.25: Reduction in indirect branch mispredictions from replicated instructions chosen using dynamic frequency in other programs.

6.6 Superinstructions vs Replication

Previous work with GForth [EG03a], an interpreter-based implementation of the Forth language, has suggested that a VM enhanced with a mixture of replication and superinstructions is more optimal than a VM enhanced with replications or a VM enhanced with superinstructions alone.

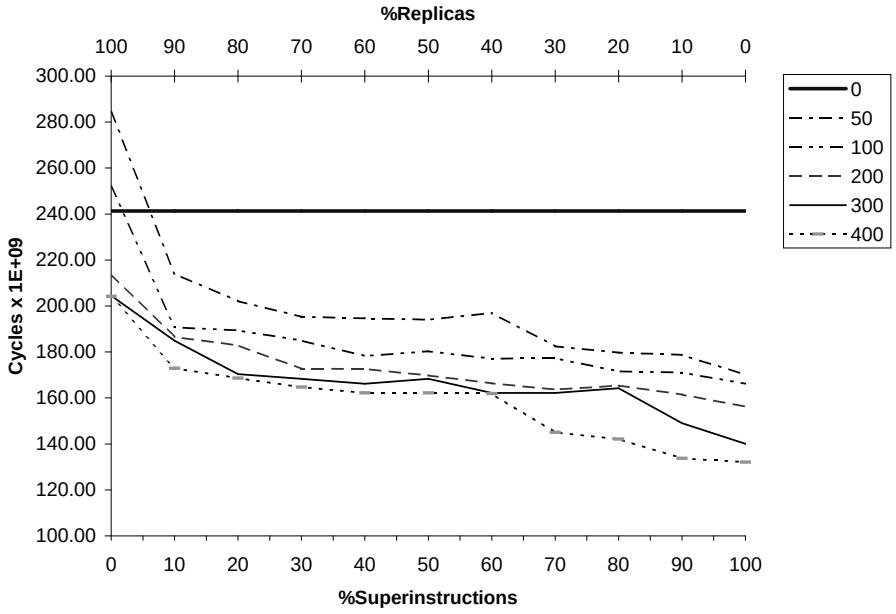


Figure 6.26: Timing results for *mpegaudio* with static replications and superinstructions on a P4; the line labels specify the total number of additional VM instructions

We examined the possibility of an optimal mix of static replications and static superinstructions in our interpreter. In order to do this, we ran a series of six tests using the SPECjvm98 benchmarks. For each of the six series, we fixed the number of instructions added to the VM to be 0, 50, 100, 200, 300 and 400 respectively. Within each series, we varied the mix of added instructions from 100% replications (and 0% superinstructions) to 0% replications (and 100% superinstructions).

The results were quite different, however, to those seen in GForth. As seen in Figure 6.26, there appears to be virtually no benefit in adding replications at the expense of superinstructions. As we noted in the section on static replication, small numbers of static replicated instructions can make performance worse. We see this most clearly when only 50 replications are added to our interpreter (with no static

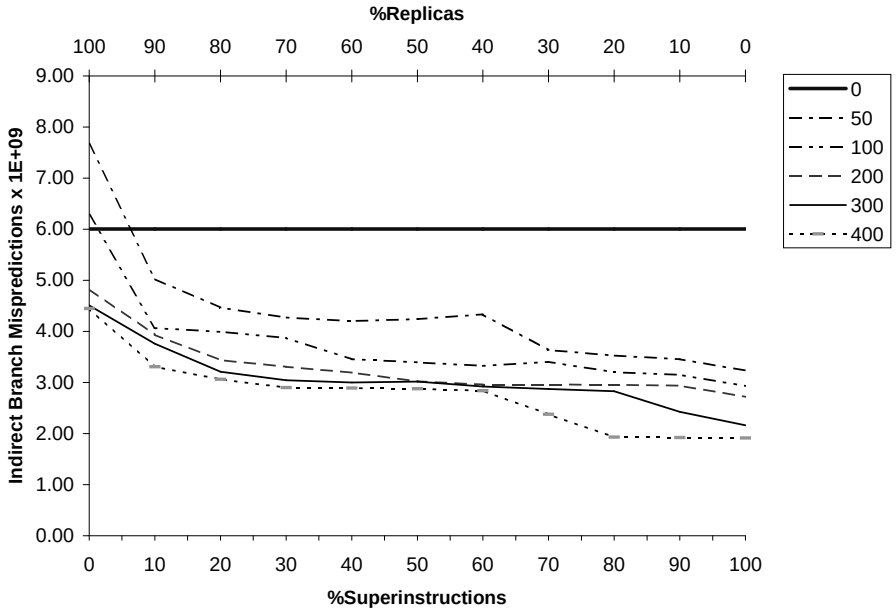


Figure 6.27: Indirect Branch Misprediction results for *mpegaudio* with static replications and superinstructions on a P4; the line labels specify the total number of additional VM instructions

superinstructions). At this point, we can see from Figure 6.27 that the number of branch mispredictions actually *increases*. The same effect is observed when only 100 replicated static instructions are added, but this time the effect is not so notable. We examined the branch prediction and branch misprediction hardware counters and at 400 replications (with no superinstructions) the misprediction rate is approximately 27%. In contrast, when using 400 superinstructions, the misprediction rate jumps to 30%. However, the superinstruction approach has only 60% of the actual branch predictions that the replication approach has, and as a result only has 66% of the number of branch mispredictions.

6.7 Conclusion

In this chapter, we have described a system of enhanced VM instructions for a portable, efficient Java interpreter. Our interpreter generator automatically creates source code for specialised instructions, superinstructions and replicated instructions from simple instruction definitions. Our system deals with several difficult issues, such as allow-

ing specialised versions of "quick" instructions, superinstructions containing "quick" instructions, and superinstructions that extend across basic blocks.

We have evaluated these VM instruction enhancement techniques using many different strategies for applying them. We found that although instruction specialisation achieves its goal of reducing operand fetch, its impact on indirect branch prediction has a much greater impact on performance. We have shown how our Tiger generator can be used to create an optimised version of our interpreter which is customised with superinstructions for a particular program, giving speedups of 1.35 to 3.35. We have also experimented with a large number of strategies for selecting useful superinstructions form a group of representative programs. Perhaps counter-intuitively, we found that it is better to look at the static occurrence of sequences of instructions rather than their dynamic execution frequencies, and that we should favour shorter sequences. Speedups of 1.2 to 2.1 are possible using such generic superinstructions. Finally, we found that instruction replication does not always lead to speedups, and that the effect on branch mispredictions is only positive for large numbers of replicas.

In the next chapter we examine dynamic instruction enhancements, an approach bridging the gap between interpreters and Just-In-Time compilers.

Chapter 7

Dynamic Instruction Enhancement

7.1 Introduction

In the previous chapter, we presented methods for statically enhancing the instruction set of the VM interpreter and translating the bytecode to take advantage of these enhancements. Because of the static nature of these enhancements, much analysis had to take place before new instructions were added. For example, when deciding which superinstructions to add to a virtual machine, it was necessary to look at the behaviour of a number of Java programs before selecting a set of superinstructions that were universally useful.

In practice, we rarely know in advance what bytecode is to be run on the interpreter. We can guess based on other programs, but the set of static instruction enhancements will rarely be optimal. In this chapter[1] we turn our attention to dynamic instruction enhancements, the creation and the use of new instructions based on bytecode that is being interpreted. No matter what bytecode is run on the interpreter, new instructions can be created on the fly. This has the advantage of guaranteeing that any new instructions created will actually be used, since their creation is based on the actual bytecode to be executed.

[1]The work presented in this chapter has been submitted to ACM Transactions on Programming Languages and Systems (TOPLAS). The paper is composed of Forth and Java VM sections. The author of this thesis implemented and benchmarked the Java VM described in this paper.

```
VM_LABEL(fload_0):
{
        SlotVal32 f1;
        {
                f1=locals[0];
                vm_StackVal32_equals_SlotVal32(SPPTR[0],f1);
        }
        IPPTR=IPPTR+1;
        SPPTR=SPPTR+1;
        VM_LABEL_END(fload_0):
        goto **IPPTR;
}
VM_LABEL_FINAL_END(fload_0):
```

Figure 7.1: Code copying labels in Tiger-generated code

7.2 Code Copying

All the approaches presented in this chapter make use of code copying techniques. Typically this involves copying code out of the existing interpreter core while the interpreter is running. The copied code then forms all or part of a new, dynamically generated instruction. Sometimes the entire instruction implementation is copied from the interpreter core into a new area, and sometimes just part of the instruction. When part of an instruction implementation is copied, it will be the entire instruction implementation minus the dispatch code.

To assist this code-copying process, Tiger generates three macros scattered around the generated code for each instruction in the interpreter core. The first of the macros (VM_LABEL(*opcode*)) identifies where the implementation code for *opcode* begins in memory. The next macro (VM_LABEL_END(*opcode*)) identifies the end of the main implementation of *opcode* and the beginning of the dispatch located at the end of that instruction's implementation. The last macro (VM_LABEL_FINAL_END(*opcode*)) identifies the end of the dispatch and thus, the end of the entire instruction's implementation. Figure 7.1 shows these labels positioned around the Tiger-generated code for fload_0[2].

We should note that code copying is dependant on GCC's labels-as-values extension.

[2]The location of the VM_LABEL_END inside the inner scope does not cause any cope-copying difficulties in our experience with GCC.

Without the extension, locating the implementation of an instruction for copying purposes at runtime in the interpreter core cannot be easily done. It is entirely possible that other less portable methods could be used, but we are only interested in highly portable optimisations. While on the subject of portability, we should also note that code copying will sometimes require a minimal amount of platform specific code, for example flushing the Instruction Cache on some processors, and on processors with no-execute (NX) bits such as AMD's Opteron and Athlon 64 processors, ensuring that the copied code has its execute flag set.

7.3 Non-Relocatable Code

Not all instructions in the interpreter core can be copied to a new area of memory and executed there. This can happen for reasons such as function calls on the x86 CPU (a PC-relative call) within the instruction implementation. The possible reasons depend on the architecture and compiler in use. Previous work [PR98] used a no-copying list to indicate which instructions could be relocated and which ones could not. However, in order to ensure portability, it is not possible to use a no-copying list since the relocatability of an instruction can change after porting the VM to another architecture. Therefore we take the approach first applied to GForth [EG03a] where two versions of the VM interpreter function are created, one of them having padding between the individual VM instructions. Then we compare the compiled code fragments of the two functions. A match indicates that we can relocate the instruction whereas a mismatch indicates that it is possible that the instruction is not relocatable.

The point of a mismatch indicating only the possibility of non-relocatability was explored further. Specifically we found that registers could be allocated differently between the two copies of the same VM instruction. This in itself did not indicate non-relocatability, just that the machine code for the two copies was different. Thus we took a second more aggressive step, after the initial automatic comparison of the two interpreter cores. We experimented with certain VM instructions that had been marked as non-relocatable but for which, on inspection, we could see no reason for non-relocatability. Allowing these instructions to be relocated and testing the resultant VM was relatively easy to do as there were only a handful of instructions misidentified as non-relocatable.

In summary, we recommend the code comparison approach to get an approximate and conservative no-copying list for a specific architecture and compiler combination. Then, if time permits, a short period of experimentation by allowing some of the more widely used and non-relocatable VM instructions to be relocated might be time well spent if even just some of those instructions turn out to be relocatable in reality. This is a relatively straightforward (and optional) step and does not increase the porting workload by any significant amount.

A significant number of the VM instructions contain jumps to labels outside the VM core. Unfortunately on the x86 platform, these jumps are relative jumps and thus, VM instructions containing these jumps cannot be relocated. To ensure these instructions could be relocated, we converted these direct jumps into indirect jumps. These indirect jumps, although slower than direct jumps, were relocatable and thus any speed losses converting jumps were more than offset by the execution speed gains resulting from the improved branch prediction rate when containing instructions were rendered relocatable. It may be a cause of concern that, when we are actually trying to reduce branch misprediction rates, we are introducing indirect branches at certain points in the VM core. However, these branches are entirely predictable, with only one branch target at each point. Secondly, all of the branches that were converted to indirect branches were infrequently executed branches that only occurred, for example, when an exception is thrown.

Some instructions also contained function calls which prevented the containing VM instruction from being relocated. We experimented with implementing the function calls through function pointers which enabled the VM instruction to be relocated. The effect of this optimisation was minimal though, mainly because the affected instructions were not commonly executed at runtime.

7.4 Dynamic Replication

In Section 6.5 we presented the technique of (static) replication. The idea of this technique was to add new copies of existing instructions to the VM interpreter and then introduce these copied instructions in a round-robin manner to the bytecode. The motivation for this approach was to reduce the branch target misprediction rate which it did successfully by a factor of 40% to 80% at 512 replications. The difficulty

with static replication is that the number of replicas must be decided in advance and the limited budget of replicas almost guarantees that a particular replicated instruction will be used more than one point in the bytecode and will thus quite likely suffer from branch mispredictions.

The dynamic equivalent of this technique involves an extra step when converting bytecode to threaded code. Previously, the technique was to place the address of the implementing instruction in the VM core into the instruction stream instead of the index for that instruction. Now, before the method translation begins, a contiguous area of memory large enough for replicated code is created. Then, each time an address for an instruction is to be written into the threaded code, the translation routine checks to see if the implementation for that instruction can be replicated (by consulting the no-copy list). If it cannot be replicated, the translation routine proceeds as normal and places the address for the implementation of that instruction in the VM core into the instruction stream. If, however, the VM instruction can be replicated, then the translation routine copies the instruction implementation from the VM core into the next available space in the area of memory reserved for replicated code. The translation routine then places the address for this copied version of the implementing VM instruction into the threaded code. Thus when the threaded code executes, it will always use the replicated versions of VM instructions when the original instructions are marked as relocatable in the no-copy list.

Figure 7.2 shows the relationship between the VM interpreter core, the threaded code and the replicated code. In the example, both VM instructions A and C are relocatable while B is not. Thus, for the threaded code, replicas are created for A, C and A again in that order in the replicated code region by copying from the VM core. Note how two replications for A, A_1 and A_2 are created in the replicated code. Also note how the threaded code instruction pointers point to implementations of the VM instructions in the replicated code region and not the VM core. This happens for all the instructions except B which is non-relocatable. The implementation for B that is used is the one in the VM core.

A final point worth noting is that there is no requirement for placing the replicated instructions in an ordered, contiguous region of memory when implementing dynamic replication. As each of the replicated instructions, A_1,A_2 and C_1 have a dispatch at the end of the replicated code, they will always jump to the address of the next instruction

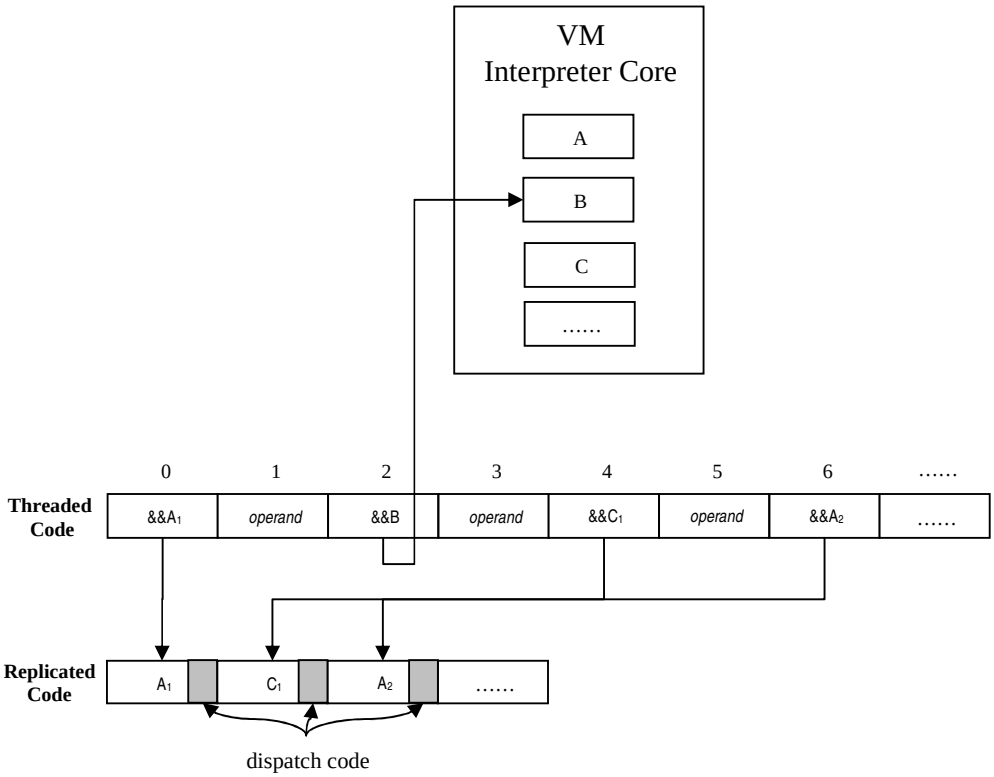


Figure 7.2: Code Replication with Relocatable and Non-relocatable VM Instructions

listed in the threaded code. Therefore it is feasible to place each of the replicated instructions in different parts of memory. However placing the replicated instructions in an ordered manner (i.e. in the manner they appear in the bytecode) and in a contiguous manner has some advantages:

1. Allocating memory is a relatively computationally expensive task. It is better to allocate one contiguous chunk of memory for a method rather than multiple small chunks for each of the replicated instructions.

2. Placing all the replicated instructions close to each other and in the same order increases code-locality. Instruction cache performance should increase as a result.

3. Such placement also allows a natural extension from dynamic replication to dynamic superinstructions. As we will see in Section 7.5, some of the dispatches in the replicated code can be removed if the instructions have been replicated in the correct order.

The point about improving instruction cache performance is important. Replication, as we will see from results later, adversely affects instruction cache performance and anything that can ameliorate this should be done, such as placing replications in a contiguous area of memory.

7.4.1 Quickable Instructions

For each of the dynamic methods of instruction enhancement, quickable instructions presented us with a challenge. Since the entire method would typically be replicated the first time that method is run, it is almost certain that many instructions in the method would still be quickable instructions. We have little choice during the replication stage but to consider quickable instructions as non-relocatable instructions.

However, the question arises as to what we ought to do when a quickable instruction is finally quickened. We have to consider some options.

1. Do not ever allow quickened instructions to be replicated. This is the easiest option in terms of implementation. The downside of this approach is that quickened instructions tend to get executed quite frequently. For example, by preventing

them from being replicated, we are not replicating as many instructions as we would like.

2. When an instruction is quickened, replicate it into a new area of memory and put the address of that replica into the threaded code instead of the address of the implementation in the VM core. This approach would work in practice, but performance would not be optimal. The reason is that, if a new chunk of memory is allocated to hold the new replica then this new replica will almost certainly not be close enough to the replicas of adjacent bytecode to avoid instruction cache misses. In addition, each new quickening and subsequent replication would require a relatively expensive malloc call. Of course, the overhead of calls to malloc could be removed by allocating a large chunk of memory early on for replication of quickened instructions at a later stage. This might also help instruction cache performance a little.

   In the end it was decided not to take this approach, because of the instruction cache issue and also that it represented a poor developmental choice when dynamic superinstructions (Section 7.5) had yet to be implemented.

3. During the initial replication step, when a quickable instruction is encountered, do not replicate it, but ensure that a gap (a *quick-replication gap*) is left in the code for the quickened instruction to be replicated into later. Then, if the non-quick instruction is quickened later, a replication of the quickened instruction can be placed directly into this gap.

It was the latter approach that we chose in the end, because it allowed us to replicate quickened instructions fast and in a manner that would not adversely affect instruction cache performance. It did have a number of complications though. The first was that during the initial replication of a method, if a quickable instruction was encountered, we needed to record the address of the quick-replication gap left in the replicated code. This address would then be used later on when the instruction was quickened and was being replicated as the destination address for that replica.

In Section 5.4.4 we saw how a quickable structure existed for each quickable instruction. This structure could easily be retrieved, as its address was stored in the threaded code. To facilitate the storage of the address for the gap where the replica should be

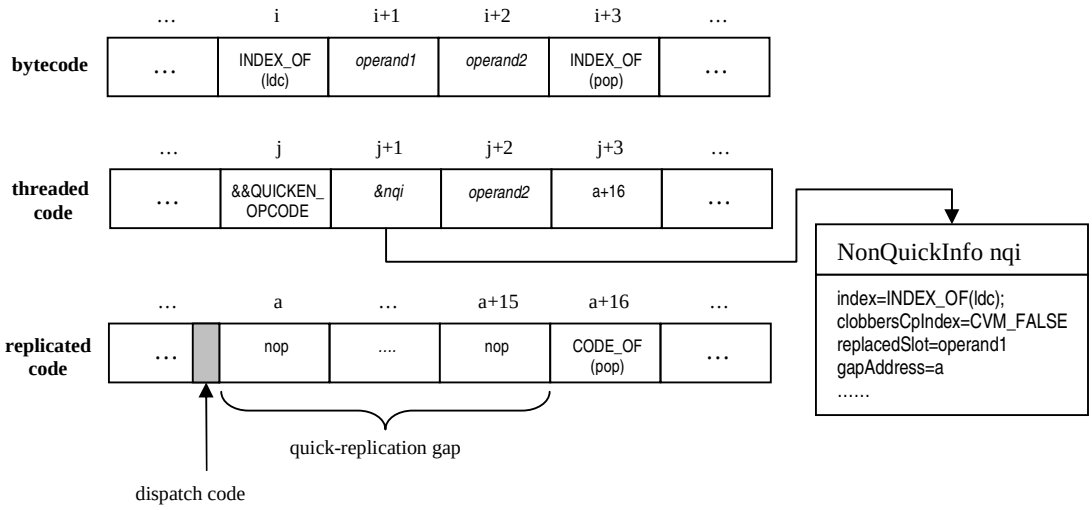
written, we introduced a new field into the structure to record the address of the corresponding quick-replication gap for that quickable instruction. During initial replication of a method, when a quickable instruction is encountered, a quick-replication gap of the appropriate size is created and the address of this gap is written into the quickable structure for that instruction. Later, when that instruction is being quickened, the address of the quick-replication gap can be retrieved and the replica can be written into that address. The address stored in the threaded code will be, of course, the address of the replicated, quickened instruction, which is the address of the quick-replication gap.

The sole complication with this approach is that it is not always possible to determine precisely the size of the quick-replication gap that should be left to accommodate a replicated quickened instruction. Certain quickable instructions can be quickened to one of several different quick instructions, all of different sizes. Furthermore, in the JVM it is not possible to determine at method translation time what these quickable instructions will quicken to. In such circumstances, the gap left to accommodate a quickened version of the quickable instruction will be a gap large enough to accommodate the *largest* possible quick instruction that it might translate to.

It is easy to see that when a quickable instruction translates to a smaller quick instruction than space was left for, some bytes might be wasted in the area of memory reserved for replicated code. Typically we found that these wasted bytes (*quick-replication waste*) numbered only a few bytes in the rare circumstances where this situation arose. Apart from a small amount of wasted memory, this is not much of a problem. There might be an increase in instruction cache misses but these ought to be negligible. Thus, for dynamic replication, this approach caused no secondary problems. In separate experimental work, we examined re-organising the dynamically replicated code in memory to eliminate these gaps but the overheads were too costly. Figure 7.3 shows the result of translating a method containing the quickable instruction `ldc` on a Pentium 4. A quick-replication gap of 16 bytes is left in the replicated code as `ldc` can be quickened to either of `ldc_quick` (16 bytes) or `aldc_ind_quick` (14 bytes)[3]. The figure also shows the result of the `ldc` instruction being quickened to `aldc_ind_quick`, the smaller of the three possible quick instructions. Note how quick-replication waste

[3]These byte sizes are inclusive of the two bytes long dispatch code. For example, the code of the `ldc_quick` instruction is 14 bytes and the dispatch 2 bytes for a total of 16 bytes.

Before quickening:



After quickening:

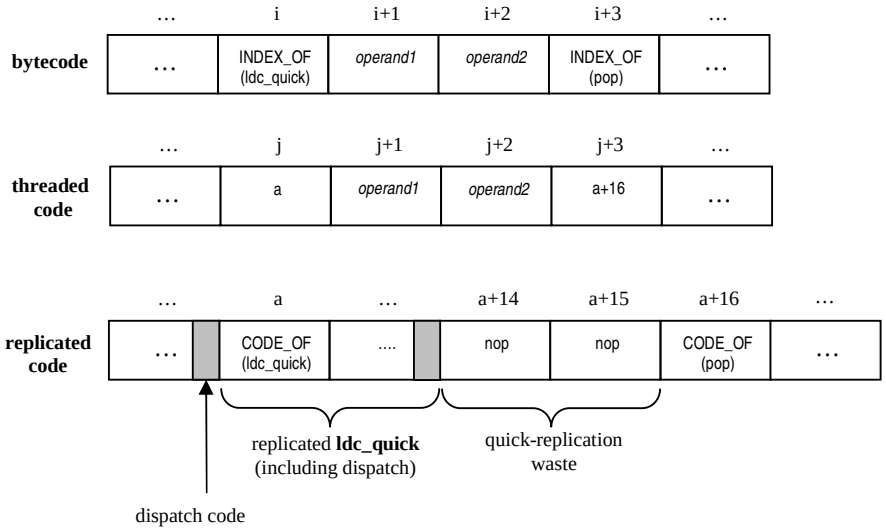


Figure 7.3: Quick Replication Gap During Dynamic Replication

space of two bytes is left in the replicated code immediately after the newly replicated quick instruction. The contents of this space will never be executed because each replicated instruction (including the newly replicated quickened instruction) contains a dispatch at the end. Thus a jump will take place before the contents of the unused space get executed.

7.4.2 Results

The results for dynamic replication are presented in Table 7.1 where we compare our optimised Fastcore interpreter from Chapter 5 against our new interpreter with Dynamic Replication enabled. The table presents data for the speedup (in terms of process cycles), percentage increase in instruction cache misses and percentage increase in branch mispredictions. We also present a fourth row, service time, which gives the estimated amount of running time spent servicing instruction cache misses. This is presented as a percentage of running time of the JVM with Dynamic Superinstructions.

From the table it can be seen that impressive speedups are to be gained from the dynamic replication approach, up to a maximum speedup of 2.49 on compress with an average speedup of 1.55 across all benchmarks. Perhaps the most telling figure is that of mispredictions. On the `compress` benchmark, only 0.38% of the mispredictions remain, i.e. we have successfully removed 99.62% of mispredictions on that benchmark. Overall, on average, only 5.79% of the mispredictions remain.

This reduction in mispredictions does come at a cost however. Examining the instruction cache figures, one can see that there is a maximum increase of 3,635% on the `mtrt` benchmark and an average increase of 483% across all benchmarks. Estimating the cost of an instruction cache miss at 27 cycles [ZR04], we calculate that 11% of running time on the `mtrt` benchmark is spent servicing instruction cache misses. Overall 5% of running time is spent on servicing instruction cache misses.

The poor performance on the `mtrt` benchmark is most likely related to its multiple-threaded nature. With dynamic replication, there is already a lot of pressure on the instruction cache and the BTB, and switching from the working set on one thread to that of another thread can affect instruction cache performance adversely and also cause conflict misses in the BTB.

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.09	1.81	2.49	1.12	1.40	1.35	0.99	1.55
mispredictions (%)	13.44	6.11	0.38	12.54	6.75	3.93	25.65	5.79
icache misses (%)	974	2000	281	615	82	276	3635	483
service time (%)	7.96	6.35	0.80	6.95	5.82	0.64	11.00	5.00

Table 7.1: *dynamic repl* performance versus Fastcore performance.

7.5 Dynamic Superinstructions

From the results presented in Table 7.1, it can be seen that the number of branch mispredictions in the dynamic replication approach is substantially reduced. The majority of the branch mispredictions that remain are from three sources. The first source of mispredictions is non-relocatable instructions. Since we are still using the single non-replicated implementation in the VM core for these instructions, each of them will typically dispatch to a wide range of address during the execution of a typical program. Unfortunately, for non-relocatable VM core instructions whose dispatch target had been reasonably predictable before, the act of replication may make the dispatch even more unpredictable. For example, if the sequence of instructions AB had occurred quite frequently, then the dispatch at A would have been relatively predictable. Supposing A was non-relocatable and B was relocatable, consider the situation after replication where occurrences of AB are replaced by AB_1,AB_2,AB_3...,AB_i where B_1,...,B_i are the various replications of B. The multiple targets of the dispatch at the end of instruction A now render that dispatch more unpredictable.

The second source for branch mispredictions is the initial misprediction inside replicated instructions, the first time they are run. Thirdly, in an ideal world, subsequent indirect branches inside replicated code will be 100% accurate once the replicated code has been executed once[4], but the initial indirect branch will always be wrong. In reality, branch prediction hardware is limited in size and so, even if a predictable branch is resolved, it may be cleared out of the prediction hardware, for example being removed from the BTB giving rise to so called conflict-misses. The more indirect branches we can remove, the more space we are reserving in prediction hardware for indirect

[4]The first time an indirect branch occurs, one is guaranteed a first time misprediction or what has been termed *cold misses* [HHR99].

branches that are really need that prediction support.

In order to remove these branch mispredictions we remove the redundant dispatches at the end of in-sequence instructions stored in the replicated code. In Section 7.4 we decided to store replicated instructions (along with their dispatches) in the same order in memory as they appear in the bytecode. The justification given at the time was to improve cache performance. However, because of the way dynamically replicated instructions are laid down in memory, we can remove some of the superfluous dispatches that take place between instructions.

By removing redundant dispatches at the end of many of the replicated instructions, we can reduce the branch misprediction rate still further by removing some of these one-time branch mispredictions. A second benefit of such a move is that by removing redundant dispatches, we are making the executed replicated code more compact, and therefore contributing to improved instruction cache performance.

Redundant dispatches can be identified as dispatches in replicated code that jump to the next instruction in the replicated code. Dispatches to non-relocatable instructions in the VM core are not redundant and therefore must be retained. To implement the removal of redundant dispatches, we used the macro-generated labels (Section 7.2) to identify where the implementation of a VM instruction began in memory and where the core of the implementation ended (but before the actual dispatch begins). Using these labels when performing the replication step, we can copy the main part of the instruction implementation omitting the redundant dispatch where necessary. When required, at the end of a dynamic superinstruction, the dispatch can still be copied from the interpreter core.

In Figure 7.4 we can see dynamic superinstruction equivalent of the dynamic replication example presented in Figure 7.2. In the example, the replication A_1 must be postfixed with dispatch code, as before since the next instruction B is non-relocatable. In the replicated code, the dispatch after C_1 can be removed since the next instruction A_2 follows it in the replicated code. (There is no dispatch at the end of A_2 because, for the purposes of the example, we assume the next instruction is relocatable).

One restriction we place on the length of the replicated dynamic superinstruction is that the dynamic superinstruction can only occur within a single basic block. This avoids any complications with control flow changes within a replicated dynamic superinstruction. Later (in Section 7.5.3) we relax this restriction and permit dynamic

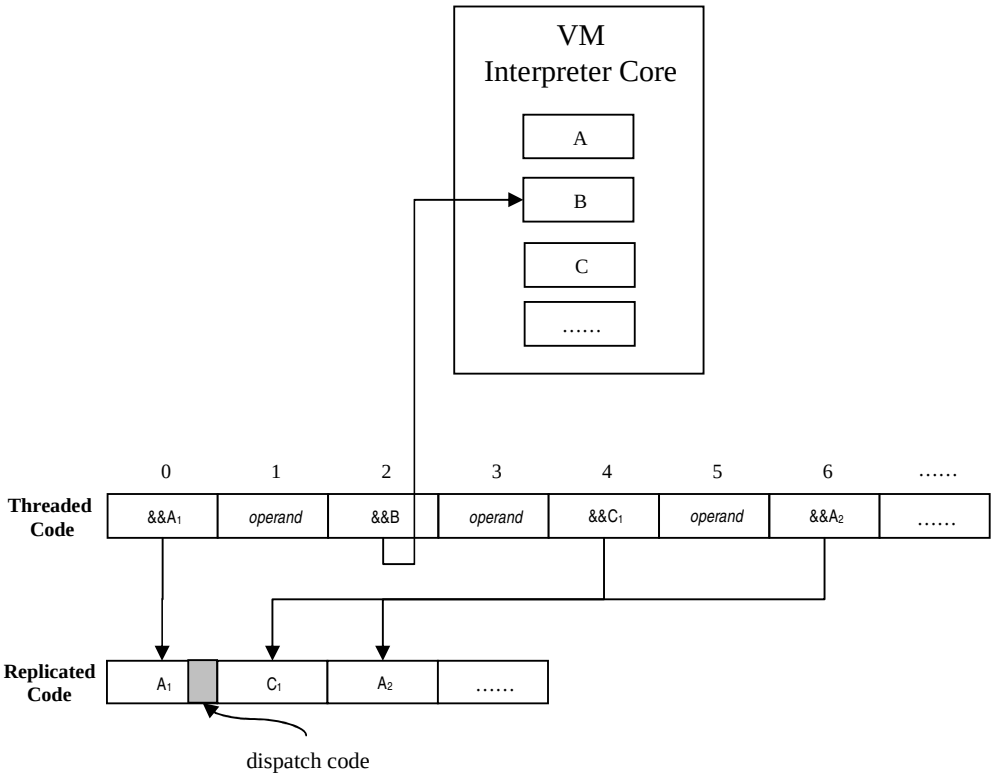


Figure 7.4: Code Replication with Relocatable and Non-relocatable VM Instructions

instructions to span basic blocks.

The net result of this optimisation is that, by simply not copying the dispatches, we are constructing dynamic superinstructions that cover sequences of bytecode up to one basic block in length.

7.5.1 Quickable Instructions

With dynamic superinstructions, as with dynamic replication, it is advantageous to allow quickened instructions be replicated. Unfortunately the method employed previously needs some modifications. When a sequence of instructions is being replicated and a quickable instruction is encountered, a *quick-replication gap* is left for the largest possible quick instruction that it might translate into. As the preceding replicated instructions no longer have dispatches, a dispatch is placed temporarily into this gap. This is effectively adding a dispatch onto the end of the preceding sequence so that, when the quickable VM instruction is encountered, a dispatch will take place to the quickening routine in the VM core. The addition of the dispatch into the quick-replication gap is the first change from dynamic replication to be made. Note that the gap left speculatively for any quickened instruction must be big enough to accommodate a dispatch. This is almost guaranteed since the dispatch code will typically be much smaller than a quick instruction implementation.

The second change relates to the small gaps (quick-replication waste) left behind in certain circumstances. As we saw with dynamic replication, sometimes when a quick instruction is replicated into the quick-replication gap, a few bytes of memory might be left unoccupied. This was not an issue with dynamic replication, since the dispatch at the end of the quickened instruction prevented the processor from trying to execute these bytes. It does, however, cause difficulties with dynamic superinstructions as this dispatch is no longer present.

A potential solution is to reorder the memory containing the quick-replication waste gap to avoid this wastage, but moving regions of memory around can be costly. Another option is to place a dispatch at the end of the replicated quickened instruction every time bytes were wasted, so that flow of control would change before the unused bytes

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.18	2.49	2.45	1.13	1.43	1.43	1.02	1.68
mispredictions (%)	8.62	0.98	0.18	7.91	6.13	3.81	22.09	3.39
icache misses (%)	568	419	16	415	68	174	2394	259
service time (%)	5.00	1.82	0.05	4.75	4.97	0.42	7.50	2.90

Table 7.2: *dynamic both* performance versus Fastcore performance.

were encountered.[5]

Instead, the approach we take is to ensure the quick-replication waste space is occupied with NOPs so that if a few bytes are wasted, the replicated code can still be executed in a straight line. Although the NOPs can cost processor cycles, this approach has the advantage of not interrupting a relatively fast straight line of dynamically replicated code.

7.5.2 Results

In Table 7.2 we compare our optimised interpreter Fastcore against our new interpreter with dynamic superinstructions with replication (*dynamic both*) enabled. As expected, the number of mispredictions is reduced when comparing against dynamic replication without superinstructions (Table 7.1). The most likely explanation for this is a reduction in capacity misses in the BTB. Dynamic replication without superinstructions involves numerous, predictable and essentially superfluous dispatches. These dispatches can flood the BTB, displacing other BTB entries, and ultimately causing capacity misses.

A sharp reduction in instruction cache misses can also be seen. The most likely effect for this is the (redundant) dispatches in dynamic replication were forcing trace-cache lines to be terminated, and thus the trace-cache being filling with numerous shortened lines. By removing some of the redundant dispatches, the trace-cache lines can be longer and instruction cache performance improves. By removing redundant dispatches from the replicated code, this code is made slightly more compact which should contribute to improved instruction cache performance. On average only 2.90%

[5]This approach was ruled out when it was recognised that it could incur serious performance penalties on some architectures. For example, the addition of an indirect branch, no matter how predictable, will result in the ending of a potential trace cache line on the Pentium 4.

of running time is spent servicing instruction cache misses on this version of the JVM versus 5.00% in the previous *dynamic-repl* version. Overall, as a result of a reduced misprediction count and a reduced instruction cache miss count, the average performance increases to a speedup of 1.68 versus a speedup of 1.55 in *dynamic-repl*.

7.5.3 Across Basic Blocks

Limiting dynamic superinstructions to within basic blocks limits the length of the average dynamic superinstruction considerably and leaves quite a number of redundant dispatches in the code. In order to eliminate these dispatches, the first step is to allow jumps into the middle of dynamically replicated superinstructions. This in itself permits longer dynamic superinstructions, but they still must still end on conditional branch instructions. For these branch instructions, we ensure that the fall-through path dispatch is at the end of these instructions and that changes to control flow get their own dispatches that occur before the end of the instruction implementation. An example of this modification at the code level was presented in Figure 5.9.

This step is similar to that of implementing the multiple dispatch points in conditional statements optimisation (presented in Section 5.5.1) and as a result it was found that all the branching instructions already been transformed in this manner. Once guaranteed that the fall-through dispatch was the one at the end of the instruction, we could remove it from the replicated code, therefore allowing the dynamic instruction to stretch across the basic block.

In Table 7.3 we compare our optimised interpreter Fastcore against our new interpreter with dynamic superinstructions with replication across basic blocks (*across bb*) enabled. In comparison to the previous version (*dynamic both*) where superinstructions were limited to one basic block. On average there are marginally fewer mispredictions, presumably as a result of fewer conflict misses in the BTB. The instruction cache performance improves by a small amount. Overall the average speedup increases to 1.74 compared to a speedup of 1.68 in *dynamic both*.

7.5.4 With Static Superinstructions

Although static superinstructions are not as universally applicable as dynamic superinstructions, they do have one distinct advantage. Because their entire code is available

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.22	2.59	2.46	1.22	1.42	1.44	1.11	1.74
mispredictions (%)	8.16	0.92	0.18	7.36	6.26	3.81	21.53	3.30
icache misses (%)	507	283	14	379	71	34	2362	236
service time (%)	4.63	1.28	0.04	4.67	5.13	0.08	8.02	2.73

Table 7.3: *across bb* performance versus Fastcore performance.

at compile time, the compiler can optimise them, making the code more compact and efficient. Dynamic superinstructions, on the other hand may not be as heavily optimised across component instructions, but they can cover most of the bytecode.

In order to examine the benefits of combining both approaches, a version *with-static-super* of the JVM was created from the version of the JVM implementing dynamic superinstructions. In this version, before replicating an instruction sequence from bytecode, the bytecode was parsed for static superinstructions. Then, static instructions would be components of the dynamic superinstruction wherever possible when replicating code during dynamic superinstruction creation.

To avoid excessive overheads, only bytecode sequences with no quickable instructions are parsed for superinstructions for inclusion in dynamic superinstructions. For bytecode sequences containing quickable instructions, a standard dynamic superinstruction without static superinstructions is created. This is quite a restrictive regime, and was relaxed so that when the number of quickable instructions in a dynamic superinstruction reached zero, the covered bytecode[6] was then parsed for static superinstructions. A new dynamic superinstruction is then created to replace the old one, containing any static superinstructions that were found. In addition, since the overhead of reconstructing the dynamic superinstruction is already being incurred, any NOPs occurring after quick instructions (i.e. quick-replication waste regions) in the old dynamic superinstruction are removed from the new dynamic superinstruction.

In Table 7.4 we compare our optimised interpreter Fastcore against our new interpreter with dynamic superinstructions with static superinstructions inside basic blocks and replication across basic blocks (*with-static-super*). Some interesting results come out of this new variation of the JVM. Firstly, the results are not as good as one might

[6]We use the term 'covered bytecode' to refer to the bytecode sequence from which the dynamic superinstruction was created.

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.10	2.60	2.69	1.18	1.38	1.37	1.10	1.72
mispredictions (%)	23.12	1.11	0.19	11.48	9.72	7.98	21.88	4.79
icache misses (%)	899	317	17	464	80	120	2313	280
service time (%)	7.38	1.45	0.05	5.54	5.57	0.28	7.81	3.21

Table 7.4: *with-static-super* performance versus Fastcore performance.

expect. Looking back to Table 7.3, it can be seen that the primary cause for decreased performance seems to be an increase in mispredictions. Oddly enough the new JVM performs better than its predecessor (*across bb*) on the compress benchmark, with virtually no noticeable increase in branch misprediction. From this we infer that the increased levels of branch misprediction are not caused solely by the addition of static superinstructions.

One possible explanation for this effect lies with the possibility that side-entries are occurring into the middle of basic blocks (i.e. the basic block boundaries are not being correctly identified under certain circumstances). Where this happens, the VM-core implementation (the non-replicated implementation), will be used for all VM instructions until the end of the basic block. The non-replicated VM-core instructions, because of heavier re-use will then contribute to an increased branch misprediction count. We are currently instrumenting our JVM to examine this behaviour in closer detail.

7.5.5 Across Basic Blocks with Static Superinstructions

A final variant of the dynamic superinstruction with replication approach mixes both the basic block spanning feature and static superinstruction incorporation into dynamic superinstructions. The main challenge with this approach emanates from the possibility that a static superinstruction which is part of a greater replicated dynamic superinstruction might lie across a branch target point. If the JVM tried to jump into the middle of that replicated static superinstruction (a side-entry), the results would be unpredictable.

A number of solutions were examined, and the preferred approach settled upon was that, when a side-entry occurred, we execute non-replicated instructions until the

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.12	2.70	2.76	1.19	1.41	1.39	1.15	1.76
mispredictions (%)	23.37	1.11	0.19	11.40	9.72	7.99	22.40	4.79
icache misses (%)	796	305	61	472	82	88	2376	279
service time (%)	6.65	1.44	0.19	5.67	5.84	0.21	8.34	3.27

Table 7.5: *with static across BB* performance versus Fastcore performance.

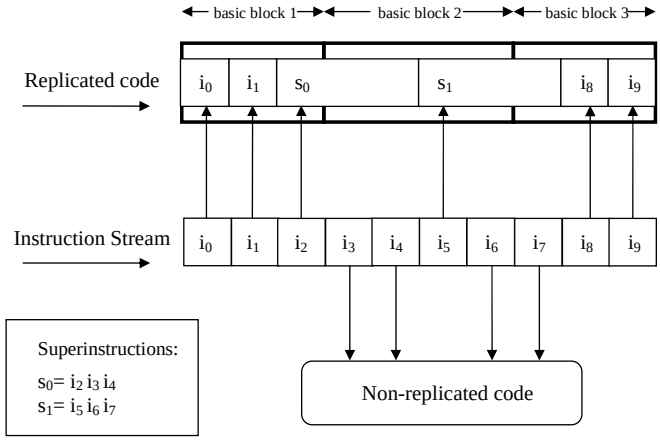


Figure 7.5: Adding static superinstructions across basic-blocks to dynamically replicated code

end of the superinstruction, and then jump back to replicated code once again. The tradeoff with this approach is that on one hand we can include many more static superinstructions in our dynamically replicated code, but on the other hand, any time a side entry occurs into a dynamically replicated static superinstruction, we revert to executing non-replicated code for the duration of that dynamically replicated static superinstruction.

An increased number of branches, and hence branch mispredictions, can result from this optimisation if side-entries occur frequently enough. In practice we found that this was not a big problem. We did experiment with a second approach for side entries, namely for each static superinstruction A_1,A_2,...,An we created a new dynamic superinstruction from the sequence of instructions A_2,...,An that we could use for side entries. However, the speed gains were negligible from this alternative method of dealing with side entries and the optimisation was dropped in order to simplify the JVM implementation.

In Table 7.5 we compare our optimised interpreter Fastcore against our new interpreter with dynamic superinstructions with static superinstructions inside basic blocks and replication across basic blocks (*with-static-super across bb*). As with the previous version (*with-static-super*), there is an elevated number of mispredictions, but despite this, the overall speedup indicates that this is the fastest of the JVMs examined so far in this chapter. It should be noted that this performance improvement is not spread very evenly across the benchmarks, but is concentrated most heavily on the `mpeg` and `compress` benchmarks.

7.6 Dynamic Superinstructions without Replication

Although dynamic superinstructions give impressive speedups, it is clear from the results that instruction cache performance is considerably worse than our reference implementation, Fastcore. This can be attributed to the sheer volume of replicated code being executed, with relatively low levels of re-use. This is similar to the inlining scheme proposed in [PR98]. The dynamic superinstruction construction method presented above can be viewed as an approach that constructs dynamic superinstructions and replicates them. This view is justified because, even if the same bytecode sequence occurs in two methods, separate copies of the replicated code will be generated for each.

In order to improve instruction cache performance, it was decided to try and increase the levels of code re-use. Each time a dynamic superinstruction was to be constructed, a hash table of existing dynamic superinstructions was consulted. If the dynamic superinstruction had not already been constructed, the code replication procedure would proceed as usual, with the additional step of adding the new dynamic superinstruction to the hash table at the end of the procedure. If the superinstruction had already been completed the pre-existing superinstruction would be re-used instead of creating it again. Thus a single dynamic superinstruction could be used by several methods or even at several places in the same method.

7.6.1 Quickable Instructions

Because several regions of bytecode can be covered by the same dynamic superinstruction, this new approach presents specific problems when it comes to quickable instructions. For example, a dynamic instruction might be created from a bytecode sequence in method A containing a quickable instruction. Later this same sequence, including the non-quick instruction is encountered in method B, so the same dynamic superinstruction is used. This is an acceptable situation, until the quickable instruction gets quickened in method A.

However, there is a problem, as we would like to modify the replicated code by writing the newly quickened instruction into the quick-replication gap. Unfortunately this dynamic superinstruction is shared by another method. If there is already a dynamic superinstruction in the JVM that matches the newly quickened bytecode sequence, that can be used instead. In such a case, the affected bytecode in method A must also be rewired to use the dynamic superinstruction containing the quickened instruction. If there is no such matching dynamic superinstruction, we have a difficulty.

The solution to this difficulty begins with the fact that our implementation employs reference counting to determine how many regions of bytecode are using a particular dynamic superinstruction. When a quickable instruction in the bytecode is to be quickened, the corresponding dynamic superinstruction, if any, is located. The reference count for this sequence is checked and if there are no other references, the quickened instruction can be written into the quick-replication gap without any difficulties since no other region of bytecode is using this dynamic superinstruction.

If there are other references, the entire dynamic superinstruction is copied to a new area of memory and the quickened instruction is written into the copied sequence. The newly quickened bytecode sequence is updated so that the new dynamic superinstruction is used instead of the old copy. Reference counts are updated as one would expect, the old dynamic superinstruction's reference count being decremented and the new dynamic superinstruction's reference count set to one.

When a dynamic superinstruction's reference count reaches zero, it is freed from memory immediately[7]. The only way a reference count for a dynamic superinstruc-

[7]It may be worth examining if this should be the policy in all circumstances. For example, it would not be desirable to free up a dynamic superinstruction, only for an identical one to be required a moment later.

tion could reach zero is during quickening. If an instruction is being quickened and the corresponding dynamic superinstruction already exists, the affected bytecode will be rewired to use the pre-existing dynamic superinstruction containing the quickened instruction. In this case, the reference count for the old quickable dynamic superinstruction will be decremented and would hit zero if the quickened bytecode sequence was the only sequence using that dynamic instruction.

It is critical to this whole process that when an instruction is being quickened, the corresponding dynamic superinstruction can be located efficiently. Thus we place enough information into the quickable structure (Section 5.4.4) for each quickable instruction in order for the corresponding dynamic superinstruction to be located. Interestingly enough, a simple pointer to the corresponding dynamic instruction is not sufficient.

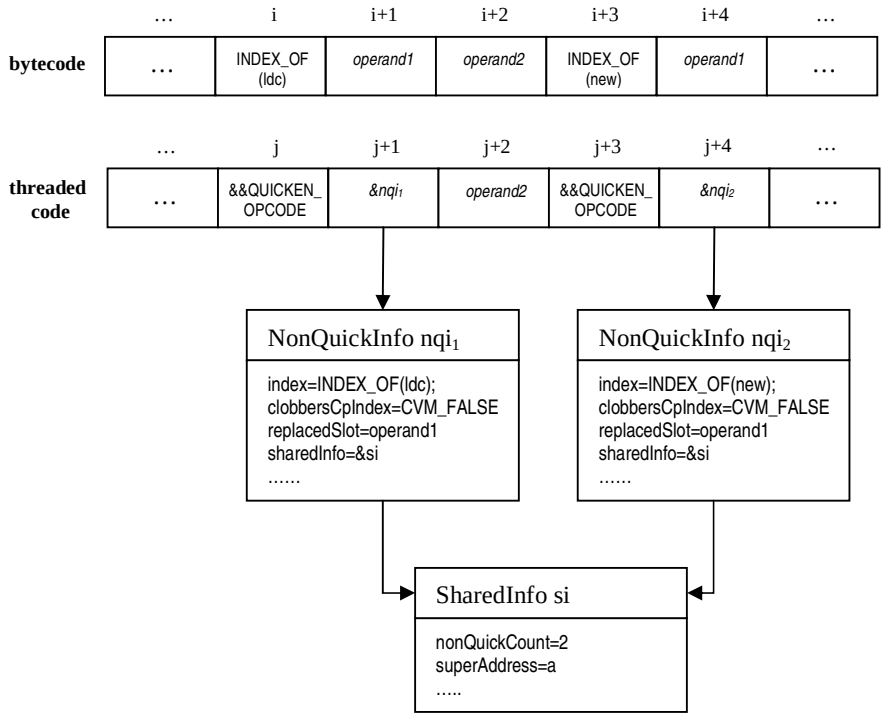


Figure 7.6: Associating a single dynamic superinstruction with multiple quickable instructions

In a region of bytecode that contains several quickable instructions, if that region is covered by the same dynamic superinstruction, all quickable instructions should allow us to find the same dynamic superinstruction. As we saw above, when a single

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.13	2.50	2.47	1.28	1.48	1.41	1.31	1.76
mispredictions (%)	16.73	2.15	0.04	12.17	15.46	5.96	20.53	4.89
icache misses (%)	764	268	120	394	43	169	529	176
service time (%)	6.47	1.17	0.34	5.09	3.21	0.41	2.12	2.06

Table 7.6: *dynamic super* performance versus Fastcore performance.

quickable instruction is quickened, the covering dynamic superinstruction for that bytecode sequence can change to be another one elsewhere in memory. All the remaining quickable instructions in that sequence of bytecode need to have quickable structures that point to this new dynamic superinstruction rather than the old one. Instead of having to visit each quickable instruction in the bytecode sequence and update the quickable structure accordingly, we take a more efficient approach. By introducing a second level of indirection as illustrated in Figure 7.6, all quickable instructions in a single bytecode sequence record their covering dynamic superinstruction in a central area (of type `SharedInfo`) and all of these quickable instructions have data that point to that common area. Then, if the covering dynamic superinstruction for those non-quick instructions changes, the pointer *superAddress* in the central area is all that is required to be changed to point to the new covering dynamic superinstruction for that area.

In Table 7.6 we compare our optimised interpreter Fastcore against our new interpreter with dynamic superinstructions with static superinstructions inside basic blocks and replication across basic blocks (*dynamic super*). One of the main justifications behind this new approach was to reduce instruction cache misses. This works quite successfully for `mtrt` which now has approximately one quarter of the misses than the dynamic methods presented previously in this chapter. However comparing, for example against *across bb* (Table 7.3), it can be seen that the instruction cache miss count actually goes up for some of the benchmarks. This is not entirely unexpected since, if a number of dynamic superinstructions located far from each other in memory are in the working set, the instruction cache can start thrashing. In cases like this, moving from dynamically replicated code to dynamic superinstructions without replication can harm code locality.

	jack	mpeg	compress	javac	jess	db	mtrt	average
speedup	1.22	2.57	2.54	1.23	1.42	1.42	1.13	1.74
mispredictions (%)	8.82	0.98	0.18	7.64	7.13	3.95	21.60	3.42
icache misses (%)	522	276	17	363	71	146	2209	234
service time (%)	4.76	1.24	0.05	4.51	5.15	0.35	7.63	2.72

Table 7.7: *dynamic super across bb* performance versus Fastcore performance.

7.6.2 Across Basic Blocks

As part of our experimental work, we examined the possibility of dynamic superinstructions that extend across basic block boundaries, but are not replicated. Our standard implementation restricts this type of dynamic superinstruction to basic blocks, a scheme identical to that presented by Piumarta et al [PR98]. The main motivation for limiting dynamic superinstructions to the basic block level is that, as longer dynamic superinstructions are permitted, there is less reuse of dynamic superinstructions. Therefore the behaviour of dynamic superinstructions across basic blocks without replication starts to approach that of dynamic superinstructions with replication as longer and longer superinstructions are permitted.

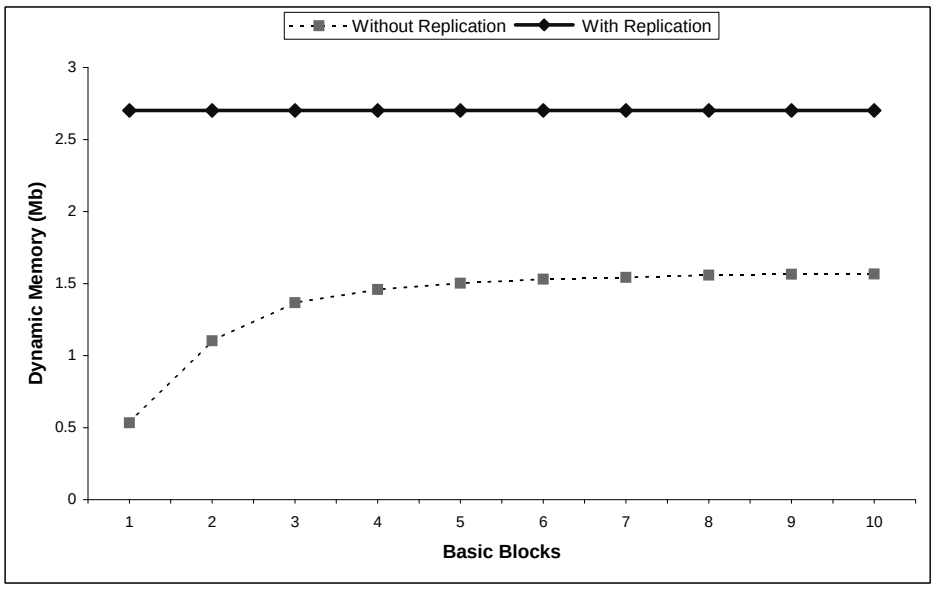


Figure 7.7: Varying the length of dynamic superinstructions without replication

This can be seen in Figure 7.7 where the average memory consumption for dynamic superinstructions across all the SPEC benchmarks is presented. The x-axis represents the number of basic blocks a dynamic superinstruction was allowed span. In the Figure, it can be seen that, as dynamic superinstructions are permitted to span more and more basic blocks, the memory requirements for the dynamically generated code increases. This indicates a sharp reduction in dynamic superinstruction reuse. The solid horizontal line in the graph represents the memory requirements for dynamic superinstructions with replication presented for comparison[8].

Even if dynamic superinstructions across basic blocks were to be permitted to stretch across an arbitrary number of basic blocks, the memory requirements will not approach those required by the dynamic superinstructions with replication approach. This is due to the fact that most dynamic superinstructions can only extend across a limited number of instructions before a non-relocatable instruction is encountered, and the superinstruction must be ended. Thus, in practice, there are limits on dynamic superinstruction size, and therefore a virtual guarantee that a certain level of code sharing (and thus memory savings) will occur. This effect can be seen in Figure 7.7, as the memory usage when dynamic superinstructions are permitted across basic blocks levels off quite early, never approaching the memory usage of dynamic superinstructions with replication.

Interestingly, experiments indicate little appreciable differences between these various versions of the JVM in terms of execution speed, even as the memory requirements changed noticeably.

Further work might examine the possibility of using fragments of longer dynamic superinstructions as the basis for shorter dynamic superinstructions. For example if the dynamic superinstruction composed of the sequence $ABCD$ existed, it could be used as a dynamic superinstruction for the sequences BCD and CD as well, although with different entry points into the dynamic superinstruction. This approach would increase dynamic superinstruction reuse and the ramp in memory consumption as we allow dynamic superinstructions without replication to stretch across basic blocks would not be so pronounced. As a direct result of the inevitable increased level of code

[8]The data presented for dynamic superinstructions with replication represents the total dynamic memory allocated for dynamic superinstructions with replication, with no limits on size in terms of basic blocks.

re-use, it should also be helpful in reducing instruction cache misses.

Looking at the actual runtime data for the *dynamic super across bb* approach in Table 7.7 we can see that allowing these superinstructions to stretch across basic blocks yields fewer mispredictions. This is no surprise since, as a result of superinstructions being longer, there will be fewer dispatches and therefore fewer mispredictions. Instruction cache performance does not degrade in comparison to dynamic super except for the extreme case of `mtrt`. The instruction cache misses for `mtrt` quadruple by allowing superinstructions to stretch across an arbitrary number of basic blocks. Ignoring the `mtrt` results, it appears that this approach gives the best broad ranging performance across all benchmarks than any of the other dynamic methods.

7.7 Conclusion

In this chapter we have examined a number of optimisations involving code replication. In all cases, porting these methods to new architectures will require very little extra code modification. Figure 7.8 shows a summary of the speedups of the various optimisations presented in this chapter and those in Chapter 6. The speed benefits of some of these methods are quite impressive with speedups of up to 2.76 on the `compress` benchmark (using the *with-static-super* variant) over our optimised JVM Fastcore interpreter.

The dynamic speedups presented in this chapter are more universally applicable than the static methods presented in the previous chapter. Figures 7.9 and 7.10 show the performance counter results of both static and dynamic methods on the `mpeg` and `compress` benchmarks respectively. All items are scaled for clarity. Miss cycles are based on an estimated cost of 27 cycles per instruction cache miss [ZR04].

Apart from the speedups (cycles), there are a number of interesting features:

1. The VM with fewest executed real machine instructions is *static super*. This is due to the fact that a lot of code is presented to the compiler at compile-time and many instructions can be optimised away.

2. The number of indirect branches in the dynamic methods is extremely low compared to in the static methods (even compared to *static super*).

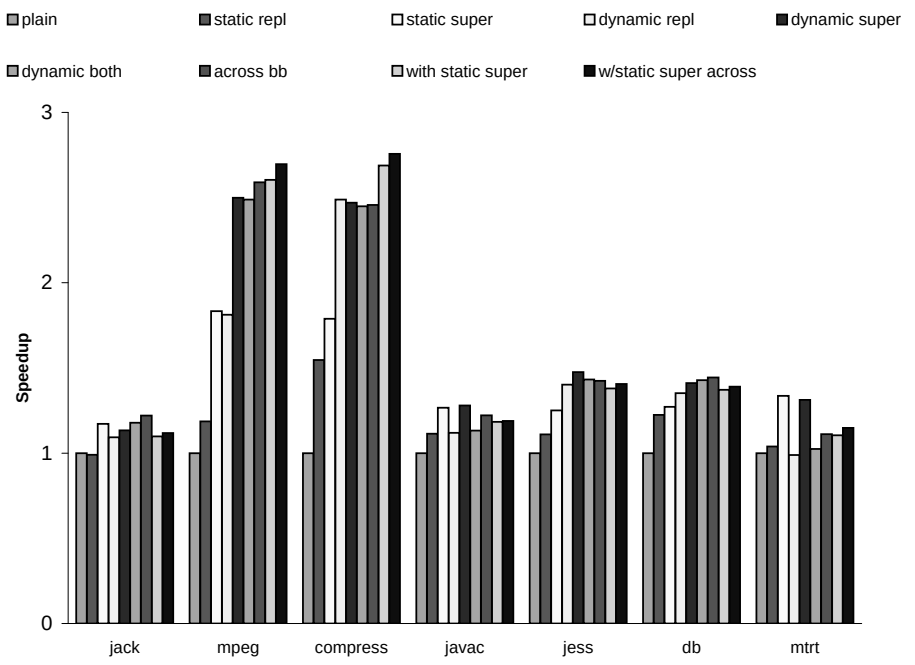


Figure 7.8: Speedups of various interpreter optimisations on a P4

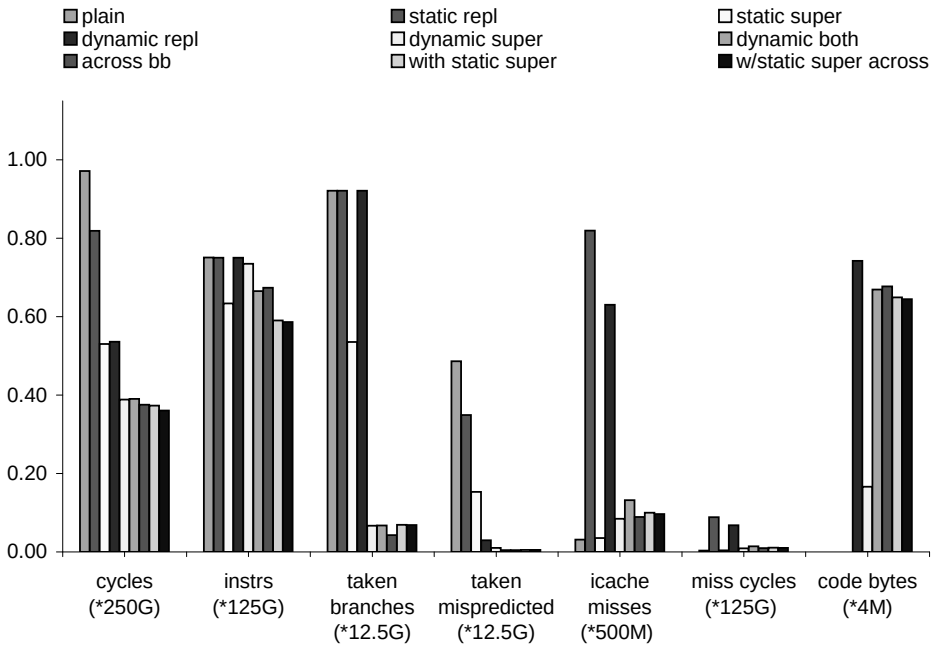


Figure 7.9: Performance counter results for *mpegaudio* on a P4

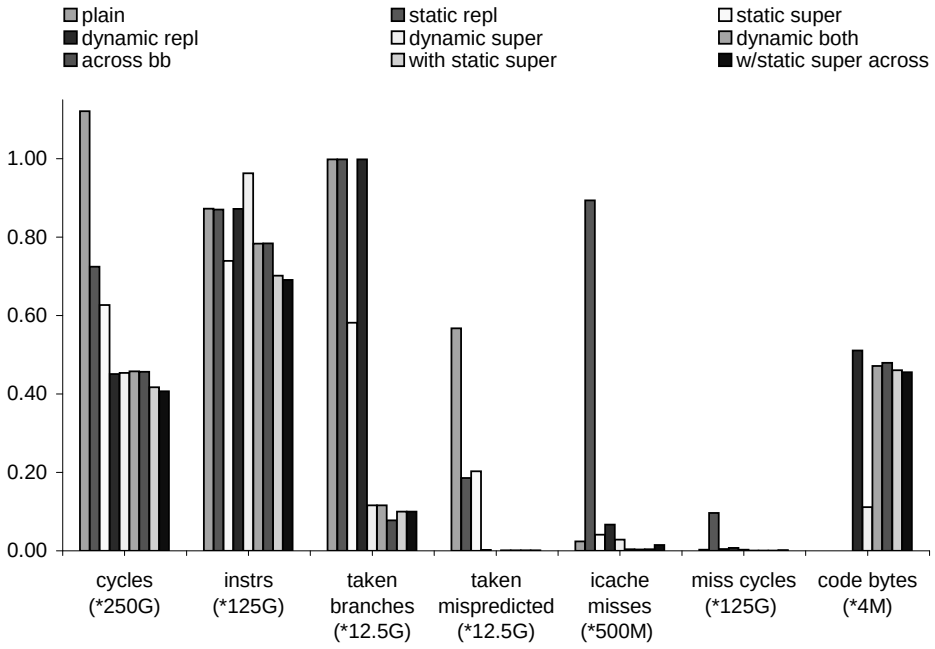


Figure 7.10: Performance counter results for *compress* on a P4

3. The number of instruction cache misses for dynamic replication is quite high, but nowhere near the levels seen in static replication (*static repl*) which has serious problems, almost certainly related to instruction cache thrashing.

4. The memory savings of *dynamic super* (dynamic superinstructions without replication) in comparison to other dynamic methods are substantial.

The main cost of these methods is in terms of dynamic memory requirements to accommodate the replicated code at runtime. There is an associated increase in instruction cache misses as a larger body of code is now being executed with less re-use. We have seen how using dynamic superinstructions without replication can help ameliorate both problems. Future work should concentrate on increasing the level of dynamic superinstruction code re-use by allowing suffixes of existing dynamic superinstruction be used as shorter dynamic superinstructions where necessary. This should reduce dynamic memory requirements and instruction cache misses yet further. Another, perhaps complementary approach, may move superinstructions which are called in quick succession closer together in memory to improve instruction cache performance yet again.

benchmark	Hotspot (mixed mode)	dynamic super	across bb	w/static across bb
jack	2.53	0.50	3.08	2.91
mpeg	0.32	0.67	2.71	2.58
compress	0.34	0.45	1.92	1.82
javac	2.63	0.80	4.42	4.22
jess	1.14	0.56	2.76	2.63
db	0.32	0.45	1.99	1.89
mtrt	0.74	0.51	2.40	2.29

Table 7.8: Peak dynamic memory requirements (Mb) on various benchmarks

In order to estimate the additional memory required by a JIT compiler, we used the memusage tool (from the glibc-utils package) to examine peak heap size in Hotspot's interpreter mode. We then ran the same VM in mixed mode and measured the increase in peak heap size. This gives us an estimate of the additional memory requirements of the mixed-mode over the interpreter mode. These results are presented in Table 7.8. Although the Hotspot results should be treated with some caution due to the nature of estimation, it does appear that at the very least, dynamic super appears to be competitive with the Hotspot VM in terms of memory requirements. Bearing in mind that inlined code reuse in dynamic super is almost certainly higher than that of the Hotspot VM (in mixed mode), this result is expected. At the same time, the Hotspot VM only invokes the JIT on more commonly used methods, so it is no real surprise that *across bb* and *with static super across bb* have much higher memory requirements, since they inline all methods.

Comparing our dynamic code copying approach to the Just-In-Time compilation approach, we observe that the amount of dynamic code our approach generates will be of the same order as that of a JVM. As pointed out in by Ertl et al [EG03a], if the JIT uses a significant amount of loop unrolling, then the JIT may actually end up consuming more memory at runtime. Substantial memory savings could be made in our interpreter by conditionally converting methods to dynamic code based on estimates of frequency of usage.

In Table 7.9 we summarise the speedups over plain using *with static across bb*, Kaffe JIT and Sun Microsystem's Hotspot JVM in both interpreter and mixed-mode. The difference between our interpreter and the native code compilers is not the orders of magnitude one might expect. Comparing the results of *with static across bb* to those

	w/static across bb	kaffe JIT	Hotspot (interpreter)	Hotspot (mixed mode)
jack	1.12	1.13	1.01	4.24
mpeg	2.70	7.52	1.07	15.14
compress	2.76	13.02	1.13	13.28
javac	1.19	1.76	1.20	5.11
jess	1.41	1.51	1.41	9.87
db	1.39	2.74	1.28	4.37
mtrt	1.15	2.16	1.02	14.52

Table 7.9: Speedups of *w/static across bb*, two native code compilers and an optimised interpreter over *plain*.

speedups obtained from Hotspot in interpreter mode shows the utility of these methods in optimising interpreters for better performance. The outperforming of Hotspot in interpreter mode is a significant achievement since it has a much faster run time system than CVM and is much less portable than our interpreter written in C.

Chapter 8

Final Thoughts

In this thesis, a number of optimisation techniques for interpreters are presented and evaluated experimentally. This chapter presents some final thoughts on the overall results and identifies some interesting possibilities for future investigation arising from this work.

8.1 The Importance of the Right Tool

Most of the work presented in this thesis was supported directly or indirectly by features of the Tiger interpreter generator. A lot of optimisations were examined which would have taken a considerable amount of time to manually code. Many of the optimisations, such as early loading (Section 4.2.6) require many small optimisations to numerous parts of the interpreter core. Tiger automates this type of process, making it substantially easier to implement. Since Tiger was developed as part of the project, there were coding overheads. The question arises as to whether that time spent developing Tiger would have been better applied to applying those optimisations by hand. However:

1. We were not quite starting from nothing. We had a good idea about the sort of functionality we wanted and how it might be supported from our experiences with Ertl's vmgen[EGKP02].

2. Providing support for an optimisation in Tiger rather than implementing it by hand in the interpreter allowed us to examine subtle variations of the optimi-

sations. These variations would have been onerous to re-code by hand in the interpreter.

3. Tiger was critical to our argument that a JVM could be optimised quickly and portably. Having to hand-code each of the optimisations would clearly take a long time. An interpreter writer with a tool like Tiger already at their disposal could spend a minimum amount of time importing their interpreter core into Tiger, making an array of optimisations suddenly possible.

4. Tiger provides profiling and debugging for standard Java instructions and any instructions added to the interpreter by any of the techniques in Chapter 6. This automatic insertion of debugging and profiling code into what were already automatically generated instructions was invaluable.

5. Tiger was designed as a general purpose interpreter generator. It is not just useful for the work completed in this thesis. We plan to use it in future work and may apply it to interpreters other than Java interpreters.

8.2 Interpreters can be Both Optimised and Portable

It is not immediately obvious that an interpreter written in 'C' with no hand tuned assembly language should be highly efficient and portable. Yet, when one factors out the one-time cost of the development of Tiger, we have proven that this is possible. Once Tiger had been constructed, creating our new interpreter core in Tiger was a relatively fast procedure. After this new core was constructed, the resulting Fastcore JVM was tested against an unoptimised token threaded version of the same virtual machine and against a number of other JVMs. Comparing like with like, the new Fastcore interpreter had an average speedup across all benchmarks of 1.31 over the unoptimised version. The highly hand-optimised non-portable HotSpot interpreter is only 20.4% quicker than the Fastcore interpreter. This is encouraging since the HotSpot interpreter contains hand-tuned assembly language and optimisations such as superinstructions and stack caching. Dispelling the myth that interpreters are *always* slower than JIT compilers, we note that in tests of Fastcore against the Kaffe JIT, the Fastcore JVM was actually faster on some benchmarks. It should be noted that

the Kaffe JIT compiler does not produce optimal code. We do not attempt to argue that *efficient* JIT compilers are usually faster than interpreters. An important point to be made in relation to this work is that the execution engine is only part, albeit an important part, of the JVM. An inefficient runtime system can cause either an interpreter or a JIT compiler to slow down considerably. We make this observation on the basis that certain programs such as *db* spend a considerable amount of time (13% in the case of *db*), not in the execution engine, but in the runtime system (Section 5.7).

8.3 Static Instruction Enhancements Can Yield Surprising results.

During the course of the work described in Chapter 6, a number of static instruction enhancements to the interpreter were examined. Of all of these optimisations, superinstructions across basic blocks performed best, giving a speedup of to 3.35 when the interpreter is customised for a particular application. When the VM instruction set is enhanced with more generic superinstructions the best speedup was 2.1. In this work on superinstructions, a method for getting superinstructions to stretch across multiple basic blocks is presented, along with a simple method for dealing with quickable instructions. In a comparison of greedy and optimal methods of parsing, we find the two methods yield similar results. This is a conclusion since confirmed by Eller [Ell05]. Contrary to previous work carried out by Ertl et al [EG03a] where an optimal mix of superinstructions and replicas was found in GForth, we find no such optimal mix for Java. In fact, in Java, superinstructions are always preferable to replicas. Reasons for this are the different superinstruction selection schemes used and differences in the basic instruction sets for GForth and Java.

The most surprising results from the work on static instruction enhancement relates instruction replication and the *instruction replication effect*. We found that instruction replication could improve the performance of the JVM with a speedup of up to just under 2.0. However, with instruction replication (and the round robin replacement scheme that was part of it) the results were unpredictable, suggesting that the scheme is quite sensitive to the patterns of instructions in the bytecode. The most interesting aspect of replication relates to what happens when small numbers of replicas are added

to the JVM. When this is done, a speedup of under 0.85 (i.e. a decrease in speed) is obtained in certain circumstances. This occurs due to an increase in the numbers of branch mispredictions, which in turn seems to be a side effect of replicating quite frequent local loads[1]. Before replication, these local load instructions are relatively predictable branch targets. Replicating the local loads reduces the overall predictability of branches, since there are now multiple possible copies of these instructions to dispatch to.

Examining the work we carried out on instruction specialisation and the similarly poor results obtained when small numbers of specialised instructions are added to the virtual machine, we came to the conclusion that specialised instructions are essentially a different form of instruction replication. The bulk of the performance improvements gained with instruction specialisation stem from the fact that these instructions are replicas (albeit slightly modified replicas) rather than the fact that these instructions eliminate operand fetching. In fact one could consider the standard Java specialised instructions (such as `iload_0`) as replicas, since their greatest effect is on branch misprediction rates.

8.4 Dynamic Code Copying

In Chapter 7, a number of code-copying optimisations were presented and compared against each other and also compared against the static methods introduced in Chapter 6. For dynamic superinstructions, the fastest methods, dynamic superinstructions without replication and dynamic superinstructions with replication, both give similar speedups of approximately 1.75.

The addition of static superinstructions to dynamic code copying methods does not currently offer any significant improvement. Our main contribution with this work is the dynamic patching technique which is an efficient way of dealing with quickened instructions. Other novel work we present in this chapter relates to the memory consumption implications of allowing superinstructions without replication to stretch across basic blocks.

In this work, we also observed that static instruction replication causes a dispro-

[1]Local loads are particularly common in Fastcore due to the merging of the `iload`, `aload` and `fload` instructions into a single, frequently occurring instruction.

portionately high number of instruction cache misses, almost certainly related to instruction cache thrashing.

8.5 Future Work

A number of issues arising from this work warrant further investigation:

1. Re-evaluation of stack caching in the light of our understanding of the *instruction replication effect*. Because stack-caching causes multiple copies of the same instructions, we believe that the effects of instruction replication provide much of the improvements we attribute to stack caching.

2. Static replication for token threaded code. Because token threaded code limits the virtual machine to a total of 256 opcodes, it is not usually possible to introduce many copies of instructions into the bytecode. However, by using multiple copies of the interpreter core much in the same way that stack caching is implemented, this limitation can be overcome. One difficulty with this approach that may need addressing is the large number of token threaded dispatch tables that may be required if a significant number of replications are introduced.

3. Re-evaluation of instruction specialisation. Since instruction specialisation is also a form of replication and is also dominated by the replication effect, we believe a re-evaluation of specialisation is warranted. We are currently preparing a careful experiment to separate the effects of specialisation from replication.

4. Optimal static instruction replication. While the round robin algorithm is an improvement of a random placement algorithm, we expect that better placement algorithms for replicas exist.

5. Dynamic Superinstructions with extensive re-use. At present dynamic superinstructions with and without replication give similar results. We believe it would be interesting to allow *extensive* re-use of dynamic superinstructions in a more extreme variation of superinstructions without replication. This can be achieved by preventing new superinstructions that are postfix subsequences of existing superinstructions from being created. Instead the postfix subsequence of the existing superinstruction would be used.

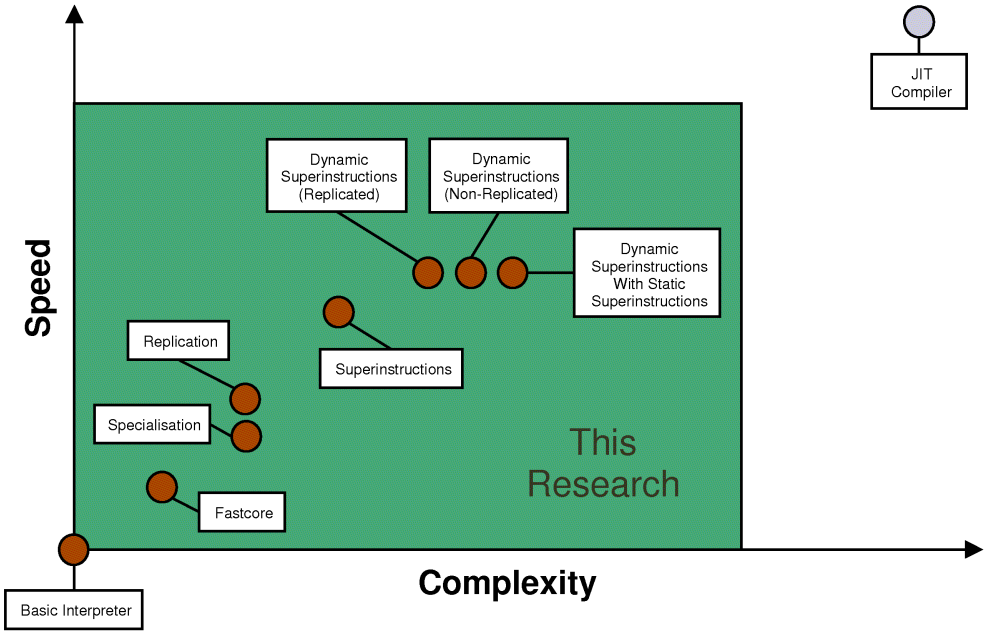


Figure 8.1: Broader context of the work in this thesis

8.6 Conclusion

In this thesis, a wide range of optimisations for VM interpreters have been explored in an attempt to narrow the gap in running speed with JIT compilation. These various optimisations are depicted in Figure 8.1 where the relative complexity of implementation versus runtime speed is plotted for these optimisations. Using such information a VM-implementor can better balance about the resources they have available versus the speed that they want their product to attain.

An important part of this work has been automating the optimisation process. Interpreter optimisations typically involve relatively small changes to large numbers of pieces of code. Traditionally, such optimisations made the interpreter increasingly unmaintainable. To overcome this problem, we have constructed Tiger, a tool that

generates an optimised interpreter from a simple, domain-specific description of the VM instruction set. Tiger performs domain-specific optimisations, and the programmer can easily experiment with different combinations. One view of these optimisations, particularly the code-copying ones is that they can provide more choices between the extremes of memory-hungry, non-portable, generally faster JIT compilers and memory-efficient, portable, generally slower interpreters.

Our results show that applying source-level optimisations such as faster dispatch methods, constant inlining, conditional loading of operands and faster method dispatch to a portable Java interpreter can result in average speedups of approximately 1.31. Static instruction enhancements such as superinstructions can result in *further* speedups of approximately 2.1 (or 3.35 if one is tailoring the interpreter for a specific application). Alternatively, more broadly applicable dynamic optimisations such as dynamic superinstructions can yield speedups of up to 2.76. The accompanying figures from the hardware performance counters underline the importance of branch prediction and, to a lesser extent, instruction cache misses on the overall performance of the interpreter.

These results give greater insight into the behaviour of VM interpreters and help the construction of simpler, faster, more maintainable VM interpreters.

Appendix

Recommended Usage for Superinstruction Parse File

The data supplied by Tiger in the automatically generated superinstruction parse file can be used in any way the interpreter-writer sees fit. However the running of a DFA is intended to occur in a certain way. The DFA starts in state 0. Each time the DFA is in state i reading symbol s, the following should happen:

1. State i is examined to see if it is a collapsed state that has yet to complete all internal transitions.

   - If the next symbol expected is s, then state i is moved to its next internal state (ie waiting for the next symbol, if any). Goto step 9.
   - If the next symbol expected is not s, then the DFA is terminated. Goto step 9.

   If state i is a collapsed state that has completed all its internal transitions, then it is treated as a normal state.

2. If state i is a normal state, then we use the entryPoints[] array to locate the hash-table for state i. The beginning index for this hash-table is given by entryPoints[i].

3. Using the beginning index for the hash-table for state i in step 2, the symbol currently being read is added to this beginning index to calculate a new index j.

4. A bounds check is performed to make sure j is a valid index for the sharedTable[] array. In order for j to be in-bounds, it needs to be non-negative and less than VM_SHAREDTABLESIZE. If j is out of bounds, end the DFA and goto step 9.

5. The pair of integers stored at sharedTable[j] is examined. The first integer of the pair indicates the owning state of this entry (i.e. which state's hashtable this entry belongs to). This integer is examined to ensure this entry belongs to the hash-table for this state. If it does not, end the DFA and goto step 9.

6. The second integer in the pair indicates the new state to enter into. If the new state is 0, then this entry in the hash-table is considered to be blank (one cannot re-enter the start state which is state 0). If this second integer is 0, end the DFA and goto step 9.

7. If the new state to enter is non-zero, we have successfully found an entry corresponding to the current state-symbol pair. If the new state is positive, this new state is entered and we goto step 9.

8. If the new state is negative, this marks the fact that it is a final state and a superinstruction should be emitted. In order to do this, the new state is converted to an absolute value. A lookup into the vm_codes[] array using this absolute value gives the index of the superinstruction to be emitted at this point. The DFA remains in this state even when the superinstruction is emitted. This is because of the possibility that the superinstruction just emitted could be just a suffix of a longer superinstruction that may be parsed.

9. End of symbol-handling.

In practice, when a method is being scanned, multiple DFAs may be running at any moment (one for each potential superinstruction at that moment). In order to put an upper bound on this number, VM_MAX_LEN can be used. This value represents the length of the longest superinstruction and therefore gives an upper bound on the number of DFAs running at any point. A tighter upper bound could exist, but Tiger does not currently attempt to find one. Once more, it must be noted that Tiger only provides the data structures to facilitate the running of these DFAs. It is up to the interpreter-writer to keep track of the current state (and any internal state) of any running DFAs. This is a relatively straightforward matter of maintaining two arrays, one for the state of a DFA and another for the internal state of a DFA.

Shared Hashes in Tiger

In order to improve the memory compactness of the hash-tables used for superinstruction parsing, Tiger overlays several hash-tables on top of each other in the same array. In order to do this, a large array sharedTable[] is created. Each entry of this array is an ordered pair {i,j} where i is the owner state (the state to which this hash-table belongs) and j is the entry in the hash-table for that state. Because the hash-table for state i could be located anywhere in the sharedTable[] array, an offsets array entryPoints[] is required. When the hash-table for state i is required, the value entryPoints[i] indicates where it begins in the sharedTable[].

In order to determine is the hash-table for a new state i can be overlaid onto the sharedTable, Tiger attempts to find a suitable offset (which will be put into entryPoints[i]). An offset is suitable if all the entries in the hash-table for i can be stored from that offset without over-writing any non-zero entries, belonging to the hash for another state. In other words, Tiger attempts to find a point where it can put the new hash table so that no entries for a previous hash are overwritten.

Note that, because of the shared nature of the hash-table, it is no longer a perfect hash. Specifically, if we look up the hash for state i with symbol s, the entry {x,y} at sharedTable[entryPoints[i]+s] must be examined to ensure that x=i.

There are many possible algorithms and permutations for overlaying the various hash-tables on top of each other. At present Tiger ranks individual hashes according to their width. For a particular state i, this is defined as $s_{max} - s_{min}$, where s_{max} is the maximum symbol (i.e. the most positive index) for which a transition exists and s_{min} is the minimum symbol (i.e. the least positive index) for which a transition exists. Tiger then places the widest hash-table into the sharedTable, then the next widest, and so on until all hash-tables have been overlaid. The sharedTable is expanded dynamically, as required. Other algorithms may give a more optimal fit, but this algorithm appears to work well in practice (an optimal fit would be the smallest possible sharedTable).

Bibliography

[AHKR00] Matthew Arnold, Michael Hsiao, Ulrich Kremer, and Barbara G. Ryder. Instruction scheduling in the presence of Java's runtime exceptions. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 18–34, London, UK, 2000. Springer-Verlag.

[Als95] Stewart Alsop. Column: Distributed thinking — HotJava could be really hot. *InfoWorld*, 17(22), May 1995. `http://www.javaworld.com/javaworld/jw-03-1996/idgns.java.1995/idgns.java.1995.009.html`.

[Ano95] Anonymous. Netscape Navigator will incorporate Sun's Java programming language. *InfoWorld*, 17(22):16, May 1995. `http://www.infoworld.com/`.

[Arm98] Eric Armstrong. Hotspot: A new breed of virtual machine. *JavaWorld*, March 1998. `http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html`.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bar83] Joe Barnhart. Forth and the Motorola 68000. *Dr. Dobb's Journal of Software Tools*, 8(9):18–20, 22, 24–26, September 1983.

[BBG+00] Gregory Bollella, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[BCGN03] Andrew Beatty, Kevin Casey, David Gregg, and Andrew Nisbet. An optimized Java interpreter for connected devices and embedded systems. In *SAC '03: Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 692–697, Melbourne, Florida, 2003. ACM Press.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text compression.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[BDGW02] S. Byrne, C. Daly, D. Gregg, and J. Waldron. Dynamic analysis of the Java virtual machine method invocation architecture. In *2nd International Conference on Instrumentation, Measurement, Control, Circuits and Systems 2002 (IMCCAS02)*, pages 1611–1616, Cancun, Mexico, May 2002.

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

[Bev87] D I Bevan. Distributed garbage collection using reference counting. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 176–187, Eindhoven, The Netherlands, 1987. Springer-Verlag.

[BGCS82] Jeffrey Barth, R. Steven Glanville, Randy Clark, and Stan Stringfellow. UCSD p-System FORTRAN version IV.0. IBM Corporation, 1982. (Software).

[Bla77] Russell P. Blake. Exploring a stack architecture. *Computer*, 10(5):30–39, May 1977.

[BSW+99] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. In *JAVA '99: Proceedings of the ACM 1999 Conference on Java Grande*, pages 81–88, San Francisco, California, United States, 1999. ACM Press.

[BVZB05] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique

for virtual machine interpreters. In *Proceedings of CGO-'05*, Third International Symposium on Code Generation and Optimization (CGO), pages 15–26, San Jose, California, March 2005.

[CGE05a] K. Casey, D. Gregg, and M. A. Ertl. Optimisations for a Java interpreter using instruction set enhancement. Technical report, University of Dublin, Trinity College., 2005.

[CGE05b] Kevin Casey, David Gregg, and M. Anton Ertl. Tiger - an interpreter generation tool. In *Proceedings of Compiler Construction, 14th International Conference, CC 2005*, volume 3443 of *Lecture Notes in Computer Science*, pages 246–249, Edinburgh, Scotland, 2005. Springer.

[CGEN03] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet. Towards superinstructions for Java interpreters. In *Proceedings of the 7th International Workshoop on Software and Compilers for Embedded Systems (SCOPES 03)*, volume 2826 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2003.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[CMC+91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *The* 18^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 266–275, Toronto, 1991.

[CMW+94] William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Sadun Anik, and Wen-mei W. Hwu. Profile-assisted instruction scheduling. *Int. J. Parallel Program.*, 22(2):151–181, 1994.

[Cos99] V. Santos Costa. Optimising bytecode emulation for Prolog. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 261–277, London, UK, 1999. Springer-Verlag.

[Cre81] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), September 1981.

[Cur93a] Charles Curley. Optimizing considerations (life in the FastForth lane). *Forth Dimensons*, pages 6–12, January-February 1993.

[Cur93b] Charles Curley. Optimizing FastForth: Optimizing in a BSR/JSR ThreadedForth. *Forth Dimensions*, pages 21–26, March-April 1993.

[DBC+03] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49, San Diego, California, 2003. ACM Press.

[Dew75] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM (CACM)*, 18(6):330–331, June 1975.

[DH98] Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 167–178, Barcelona, Spain, 1998. IEEE Computer Society.

[DHPW01] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum Benchmark suite. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 106–115, Palo Alto, California, United States, June 2001. ACM Press.

[DKP00] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. *ACM SIGPLAN Notices*, 35(5):274–284, 2000.

[DM82] David R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Symposium on Architectural Support for Programming Languages and Systems*, 1982 Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS '82), pages 48–56, Palo Alto, California, March 1982.

[DS84] L. Peter Deutsch and Allan 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, January 1984.

[DV90] Eddy H. Debaere and Jan M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.

[ECM02] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag.

[EG03a] M. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), pages 278–288, May 2003.

[EG03b] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, vol. 5, November 2003.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz — open source graph drawing tools. *Lecture Notes in Computer Science*, 2265, 2002.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — A generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Ell99] H. Eller. *Threaded Code and Quick Instructions for Kaffe*. December 1999. `http://www.complang.tuwien.ac.at/java/kaffe-threaded/`.

[Ell05] Helmut Eller. Optimizing interpreters with superinstructions. Diplomarbeit, TU Wien, 2005. `http://www.complang.tuwien.ac.at/Diplomarbeiten/eller05.ps.gz`.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 Conference Proceedings*, Mariánské Láznè (Marienbad), Czech Republic, October 1993.

[Ert94] M. Anton Ertl. Stack caching for interpreters. In *EuroForth '94 Conference Proceedings*, pages 3–12, Winchester, UK, 1994.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, La Jolla, California, June 1995.

[Ert96] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, Austria, 1996.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Fog01] Agner Fog. Branch prediction in the Pentium family. *Dr. Dobb's Microprocessor Resources*, 2001. `http://www.x86.org/articles/branch/branchprediction.htm`.

[Fur88] Borivoje Furht. A RISC architecture with two-size, overlapping register windows. *IEEE Micro*, 8(2):67–80, 1988.

[GEK01] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient Java interpreter. In *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 613–620, London, UK, 2001. Springer-Verlag.

[GEW01] David Gregg, M. Anton Ertl, and John Waldron. The common case in Forth programs. In *EuroForth 2001 Conference Proceedings*, pages 63–70, Dagstuhl, Germany, November 2001.

[GH98] E. Gagnon and L. Hendren. SableCC – an object-oriented compiler framework. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'98)*, TOOLS 1998, pages 140–154, Beijing, China, 1998.

[GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–39, Monterey, California, USA, April 2001.

[GH03] Etienne Gagnon and Laurie J. Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Proceedings of Compiler Construction, 12th International Conference, CC 2003*, Lecture Notes in Computer Science, pages 170–184, volume 2622. Springer, 2003.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[GNU03] GNU. GCJ: The GNU compiler for Java. April 2003. `http://gcc.gnu.org/java/`.

[GRA+03] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–59, May 2003.

[Gri98] D. Griswold. The Java Hotspot virtual machine architecture. Sun Microsystems White Paper, 1998.

[Gri99] Robert Griesemer. Interpreter generation and implementation utilizing interpreter states and register caching. US Patent 6,192,516 B1, Feb. 20, 2001, April 1999.

[GSaC05] Paul Griffin, Witawas Srisa-an, and J. Morris Chang. An energy efficient garbage collector for Java embedded devices. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 230–238, Chicago, Illinois, USA, 2005. ACM Press.

[GW02] David Gregg and John Waldron. Primitive sequences in general purpose Forth programs. In *Proceedings of 18th EuroForth European Conference on Forth*, pages 24–32, Vienna, Austria, September 2002.

[HA00] Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java virtual machine interpreters. In *Proceedings of the 9th International Conference on Compiler Construction*, pages 35–49, Berlin, Germany, March 2000. Springer-Verlag.

[HATW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik Van De Wiel. A code compression system based on pipelined interpreters. *Softw. Pract. Exper.*, 29(11):1005–2023, 1999.

[HFWZ87] John R. Hayes, Martin E. Fraeman, Robert L. Williams, and Thomas Zaremba. An architecture for the direct execution of the Forth programming language. volume 22 (10) of *Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 42–48, Palo Alto, California, October 1987.

[HHR99] R. B. Hilgendorf, G. J. Heim, and W. Rosenstiel. Evaluation of branch-prediction methods on traces from commercial applications. *IBM Journal of Research and Development*, 43(4):579–593, July 1999.

[HL89] John Hayes and Susan Lee. The architecture of the SC32 Forth engine. *Journal of Forth Application and Research*, 5(4):493–506, 1989.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufman Publishers, 3rd edition, 2003.

[HS85] Makoto Hasekawa and Yoshiharu Shigei. High-speed top-of-stack scheme for VLSI processor: A management algorithm and its analysis. In *International Symposium on Computer Archictecture (ISCA)*, pages 48–54, Boston, MA, June 1985.

[HSS80] Dennis E. Hall, Deborah K. Scherrer, and Joseph S. Sventek. A virtual operating system. *Commun. ACM*, 23(9):495–502, 1980.

[HSU+01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, page 13, February 2001.

[Hug82] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, Pittsburgh, Pennsylvania, United States, 1982. ACM Press.

[IdC05] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005. `http://www.jucs.org/jucs_11_7/the_implementation_of_lua`.

[Int04] Intel Corporation. Intel C++ compiler for Linux systems user's guide. June 2004. Release 8.1, Document Number 253254-031, `http://www.intel.com/software/products/compilers/clin/docs/ug_cpp/lin1001.htm`.

[ISS05] ISS Technology Communications. The Intel Processor roadmap for industry-standard servers. *HP Technology Brief, 6th Edition*, April 2005. `http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00164255/c00164255.pdf`.

[Kog82] Peter M. Kogge. An architectural trail to threaded-code systems. *Computer*, 15 (3):22–32, March 1982.

[Koo89] Philip J. Koopman, Jr. *Stack Computers*. Ellis Horwood Limited, 1989.

[Koo92] Philip J. Koopman, Jr. A preliminary exploration of optimized stack code generation. In *Proceedings of the 1992 Rochester Forth Conference*, University of Rochester, NY, 1992.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[Ler96] Xavier Leroy. The Objective Caml Benchmarks. INRIA, October 1996. `ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/benchmarks/objcaml.tar.gz`.

[Ler97] Xavier Leroy. *The Objective Caml System Release 1.07*. INRIA, December 1997. `http://caml.inria.fr/pub/distrib/ocaml-1.07/`.

[Lia99] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Lie95] Erica Liederman. Microsoft licenses Java. *Sunworld Online*, 12:1, December 1995. `http://www.sun.com/sunworldonline/swol-12-1995/swol-12-microsoft.html`.

[LN04] J. P. Lewis and Ulrich Neumann. Performance of Java versus C++. *Computer Graphics and Immersive Technology Lab, University of Southern California*, 2004. `http://www.idiom.com/~zilla/Computer/javaCbenchmark.html`.

[Lou03] Robert Lougher. JamVM v. 1.0.0, March 2003.
Available from `http://jamvm.sourceforge.net`.

[LS05] John Loiacono and Jonathan Schwartz. It's all about community. JavaOne Conference, San Francisco, CA, June 2005. Keynote Talk - Details at `http://java.sun.com/javaone/sf/2005/sessions/general/sun_monday.jsp`.

[MB99] Blair McGlashan and Andy Bower. The interpreter is dead (slow). isn't it? In *OOPSLA '99 Workshop: Simplicity, Performance and Portability in Virtual Machine Design*, Denver, CO, October 1999.

[ME98] Martin Maierhofer and M. Anton Ertl. Local stack allocation. In *Proceedings of Compiler Construction (CC'98)*, pages 189–203, Lisbon, Portugal, 1998. Springer LNCS 1383.

[Mye77] Glenford J. Myers. The case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6(3):7–10, 1977.

[NÖ1] Henrik Nässén. Optimizing the SICStus Prolog virtual machine instruction set. Technical Report SICS-T2001/01-SE, Intelligent Systems Laboratory Swedish Institute of Computer Science, Box 1263, S164 29 Kista, Sweden, 2001.

[NCS01] Henrik Nässén, Mats Carlsson, and Konstantinos Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pages 49–60, Florence, Italy, September 2001. ACM Press.

[Nel79] Philip A. Nelson. A comparison of PASCAL intermediate languages. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 208–213, Denver , CO, 1979. ACM Press.

[New00] Ted Neward. *Server-Based Java Programming.* Manning Publications Co., 2000. Pg. 148.

[OP04] Diarmuid O'Donoghue and James F. Power. Identifying and evaluating a generic set of superinstructions for embedded Java programs. In *Proceedings of International Conference on Embedded Systems and Applications*, pages 192–198, Las Vegas, Nevada, USA, June 2004.

[Pat95] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 67–78, La Jolla, California, June 1995.

[Pen97] Pendragon Software Corporation. Caffeinemark 3.0, 1997. `http://www.benchmarkhq.ru/cm30/info.html`.

[Por04] Chris Porthouse. High performance Java on embedded devices. White Paper, September 2004. `http://www.arm.com/pdfs/JazelleWhitePaper.pdf`.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedingf of the SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, Montreal, Canada, June 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, San Francisco, California, January 1995.

[PWL04] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code sharing among states for stack-caching interpreter. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 15–22, Washington, D.C., 2004. ACM Press.

[RCM96] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The evolution of Forth. In *History of Programming Languages*, pages 625–658. ACM Press/Addison-Wesley, 1996.

[Rei01] Fermin Reig. Annotations for portable intermediate languages. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Microprogramming Workshop (MICRO-14)*, pages 183–198, Los Alamitos, CA, USA, October 1981.

[Ric71] Martin Richards. The portability of the BCPL compiler. *Softw., Pract. Exper.*, 1(2):135–146, 1971.

[RLV+96] 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 *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159, Cambridge, Massachusetts, October 1996.

[RST04] Allison Randal, Dan Sugalski, and Leopold Tötsch. *Perl 6 and Parrot Essentials*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, second edition, 2004.

[SGBE05] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. In *VEE '05: Proceedings of*

the 1st ACM/USENIX international conference on Virtual execution environments, pages 153–163, Chicago, IL, USA, 2005. ACM Press.

[She04] John P. Shen. *Modern Processor Design : Fundamentals of Superscalar Processors (Electrical and Computer Engineering)*. McGraw-Hill, 2004.

[Sin03] Jeremy Singer. JVM versus CLR: A comparative study. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169, Kilkenny City, Ireland, June 2003.

[Sir] Emin Gn Sirer. MIPSI - MIPS Simulator. (Software) `http://www.cs.cornell.edu/People/egs/mipsi/mipsi.html`.

[SM77] Peter U. Schulthess and Eduard P. Mumprecht. Reply to the case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6(5):24–27, 1977.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[SPE98] SPEC. Spec releases SPECjvm98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release, August 1998. `http://www.spec.org/jvm98/press.html`.

[SRD96] Gerrit A. Slavenburg, Selliah Rathnam, and Henk Dijkstra. The Trimedia TM-1 PCI VLIW mediaprocessor. In *Hot chips VIII: symposium record: Stanford University, Stanford, California, August 18–20, 1996*, pages 171–178, August 1996.

[Sun00] Sun Microsystems. The K Virtual Machine - a white paper. May 2000. `http://java.sun.com/products/kvm/wp`.

[Sun01] Sun Microsystems. Connected Device Configuration (CDC) and the Foundation Profile - technical whitepaper. May 2001. `http://java.sun.com/products/cdc/wp/CDCwp.pdf`.

[Sun05a] Sun Microsystems. *Hello World(s) – From Code to Culture: A 10 Year Celebration of Java Technology*. Prentice Hall PTR, 1st edition, October 2005.

[Sun05b] Sun Microsystems. Sun Microsystems - J2ME CDC overview, October 2005. `http://java.sun.com/products/cdc/overview.html`.

[SZY00] Liu Songyan, Mao Zhigang, and Ye Yizheng. Implementation of Java Card Virtual Machine. *J. Comput. Sci. Technol.*, 15(6):591–596, 2000.

[TB02] Bill Trippe and Kate Binder. *SVG For Designers: Using Scalable Vector Graphics in Next-Generation Web Sites*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[Ven00] Bill Venner. *Inside the Java Virtual Machine*. McGraw-Hill, 2nd edition, January 2000.

[VHP01] T. VanDrunen, A. Hosking, and J. Palsberg. Reducing loads and stores in stack architectures. Online Manuscript, September 2001. `http://www.cs.ucla.edu/~palsberg/draft/vandrunen-hosking-palsberg00.pdf`.

[VMK02] K. S. Venugopal, Geetha Manjunath, and Venkatesh Krishnan. sEc: A portable interpreter optimizing technique for embedded Java virtual machines. In *Java Virtual Machine Research and Technology Symposium*, pages 127–138, San Francisco, CA, August 2002.

[VRCG+99] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 125–135, Mississauga, Ontario, Canada, November 1999. IBM Press.

[Wal99] John Waldron. Dynamic bytecode usage by object oriented Java programs. In *Proceedings of the Technology of Object-Oriented Languages and Systems 29th International Conference and Exhibition*, Nancy, France, June 7-10 1999.

[WCL01] Kenji Watanabe, Wanming Chu, and Yamin Li. Exploiting Java instruction/thread level parallelism with horizontal multithreading. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 122–129, Queensland, Australia, January 2001. IEEE Computer Society.

[Weg96] M. Wegdam. Compact code generation through custom instruction sets. Technical Report TN 417/96, Philips Research, Eindhoven, The Netherlands, December 1996.

[Wil98] T.J. Wilkinson. *KAFFE, A Virtual Machine to Run Java Code*. July 1998. `http://www.kaffe.org`.

[Woo93] Mark Woodman. A taste of the Modula-2 standard. *SIGPLAN Not.*, 28(9):15–24, 1993.

[WP97] Phil Winterbottom and Rob Pike. The design of the Inferno virtual machine. In *Proceedings of IEEE Compcon '97*, pages 241–244, San Jose, California, United States, February 1997.

[ZR04] Jingren Zhou and Kenneth A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 191–202, Paris, France, 2004. ACM Press.