Microarchitectural Techniques to Enable Efficient Java Execution

by

Ramesh Radhakrishnan, B.Tech., M.S

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2000

Microarchitectural Techniques to Enable Efficient Java Execution

Approved by Dissertation Committee:

To Poojya Sree Bhagawan, my parents and Radhika.

Acknowledgments

First and foremost I would like to thank my advisor Prof. Lizy John, for the guidance and support she has provided me the last 5 years. I was fortunate to work on topics that interested me and Prof John was always there to point out anything I had overlooked. She has been a wonderful mentor and I am proud to be her first Ph.D graduate.

I would like to thank my committee members Prof. Jim Browne, Prof. Craig Chase, Prof. Vijay Garg and Dr Ram Rajamony for their valuable insight and suggestions during the proposal and final defense, which has resulted in improving the dissertation immensely.

I would not have been able to complete this dissertation without the help of a lot of people who offered me their technical expertise. I would like to thank Vijaykrishnan Narayanan and Anand Sivasubramanium of Penn State who worked with me on many issues related to this dissertation. Vijay's enthusiasm and dedication never ceases to amaze me, from the time I knew him as a graduate student at USF. I owe Ravi Bhargava a pitcher of beer for developing supersim, helping me modify it, being present at all my practice talks, keeping the LCA machines up and running etc etc. Deepu Talla worked with me on the picoJava processor, and I thought we made a good team (maybe our basketball sessions have something to do with that). Juan Rubio developed the bytecode tracer, which was pretty robust and flexible and therefore made life a lot easier for me. I would like to thank the rest of the LCA members, past and present for providing feedback and support the past three years.

Looking back, it has been the support of my friends that make this journey appear so short. I would like to thank all my roommates, starting from my undergraduate years - Anil Nanduri, Uday, Srikanth, Dale, Jayanth, Anil Shaji and Chethan for putting up with me, literally. Thanks to members of CERC past and present for creating a pleasant working environment in the research lab. Archana believes she has been lucky for me, and that everything fell into place once she came into my life. I cannot thank her enough for the past 2 years, which was the toughest and most crucial period. Finally, I would like to thank my parents, sister and relatives (both in India and in the US) who supported me and made this possible.

RAMESH RADHAKRISHNAN

The University of Texas at Austin

August 2000

Microarchitectural Techniques to Enable Efficient Java Execution

Publication No. ____________________

Ramesh Radhakrishnan, Ph.D.
The University of Texas at Austin, 2000

Supervisor: Lizy Kurian John

The Java Virtual Machine (JVM) is the corner stone of Java technology, and its efficiency in executing Java bytecodes is crucial for the success of this technology. Interpretation, Just-In-Time (JIT) compilation, and dedicated hardware realization are well known solutions for a JVM, and previous research has proposed optimizations for each of these techniques. This dissertation consists of comprehensive research on Java runtime systems, examining architectural issues from both hardware and JVM implementation perspectives to enable efficient Java runtime system development on a wide range of platforms and execution environments. Our research shows that software translation of Java bytecodes using interpreters and JIT compilers is the biggest bottleneck in Java performance. The actual operations required for producing the program results is only a small subset of the operations that happen during Java program exe-

cution using software translation. Software emulation is easy to implement for new platforms but cannot offer a solution for fast execution of Java bytecodes. Using hardware support to tranlate the bytecodes eliminates the requirement of a software layer to emulate the bytecodes. This dissertation proposes the use of a hardware acceleration unit that works in conjunction with a standard microprocessor to improve the execution performance of Java programs. The proposed Hard-Int architecture performs the translation of Java bytecodes to native code using hardware, thus eliminating much of the overhead of software translation. Performing hardware translation in a general purpose processor is seen to improve Java execution by upto 770% over an interpreter and upto 500% over a JIT compiler. This dissertation also investigates techniques to improve performance in a dedicated Java processor. Java processors which are typically stack machines, are limited by dependencies to the stack making it difficult to achieve high-performance due to limited instruction level paralellism. To address this problem we investigate cost-effective microarchitectural techniques to exploit parallelism in stack-based embedded Java processors.

Contents

Acknowledgments **v**

Abstract **vii**

Chapter 1 Introduction **1**
1.1 The Java technology . . . 1
1.1.1 Executing Java code . . . 3
1.2 Problem Statement . . . 5
1.2.1 Limitations of Interpreters . . . 5
1.2.2 Limitations of JIT compilers . . . 6
1.2.3 Limitations of Java Processors . . . 8
1.3 Proposed Approach . . . 10
1.4 Goals of this dissertation . . . 12
1.4.1 Outline of the Dissertation . . . 14

Chapter 2 Background and Related Work **16**
2.1 Java Execution Environment . . . 16
2.1.1 Java Virtual Machine . . . 16
2.1.2 Optimizing Java performance . . . 21
2.2 Workload Characterization . . . 23
2.2.1 Object oriented workloads . . . 23
2.2.2 Java workloads . . . 25
2.3 Software Optimizations . . . 27
2.4 Language specific hardware . . . 29
2.4.1 Java processors . . . 31

Chapter 3 Characterization of Java Runtime Systems **34**
3.1 The SPECjvm98 Benchmark Suite . . . 35
3.1.1 Method Reuse . . . 36
3.1.2 Instructions Executed per Bytecode . . . 37
3.2 Architectural Issues . . . 38
3.2.1 Instruction Mix . . . 39

3.2.2 Branch Prediction 42
3.2.3 Locality and Cache Performance 44
3.2.4 Other Observations From Cache Studies 52
3.2.5 Instruction Level Parallelism 55
3.2.6 Limits of Available Parallelism 58
3.3 Architectural Enhancements 61
3.4 Summary 64

Chapter 4 Translation Support in General-Purpose Processors 67
4.1 Introduction 68
4.2 The Hard-Int Architecture 69
4.2.1 Implementation Details 71
4.3 Experimental Methodology 74
4.4 Performance Evaluation 77
4.4.1 Execution speedup 77
4.4.2 Cache performance 81
4.4.3 Performance of the translated code buffer 83
4.4.4 Emulation cost 84
4.4.5 Performance bottlenecks 85
4.5 Summary 87

Chapter 5 Exploiting Parallelism in Embedded Java Processors 90
5.1 Introduction 91
5.1.1 Contributions 92
5.1.2 Related Work 93
5.2 Background and Motivation 94
5.2.1 Decoding instructions in picoJava-II 95
5.2.2 Folding optimizations performed in picoJava-II 96
5.2.3 Critical path analysis 100
5.3 Fill unit and DB-Cache operation 103
5.3.1 Performing folding operations 106
5.4 Effectiveness of the fill unit and DB-Cache 109
5.4.1 Instruction decode rate 109
5.4.2 Folding efficiency and performance 111
5.5 In-order multiple instruction issue 114
5.5.1 Effectiveness of in-order multiple instruction issue . . . 114
5.5.2 Using stack disambiguation to exploit concurrency . . . 117
5.6 Summary 119

Chapter 6 Conclusions 122
6.1 Summary 122
6.2 Concluding Remarks 125

Bibliography 127

Vita 134

Chapter 1

Introduction

The Java Virtual Machine (JVM) forms the cornerstone of Java technology and is responsible for the portability and security of Java applications. The JVM provides a well defined framework in which Java programs are compiled for a hypothetical instruction set architecture. Java programs can be executed on any platform that implements the JVM. Java applications are executed either by interpreting to the native instructions of the platform, compiling them at runtime to machine instructions or by directly executing them on a Java processor. The execution performance of Java applications is heavily dependent on the execution mode that is used.

1.1 The Java technology

The Java technology consists of the Java programming language, the Virtual Machine and the application programming interfaces (API). The JVM is a stack machine with integrated object-oriented features that reflect the semantics of the Java programming language. It is responsible for the portability and security of Java applications. The Java Runtime Environment (JRE) specifies the functional relationship between these components and offers a reliable environment for deploying Java applications. Figure 1.1 shows the components of the Java runtime environment and the functional relationship between the components.

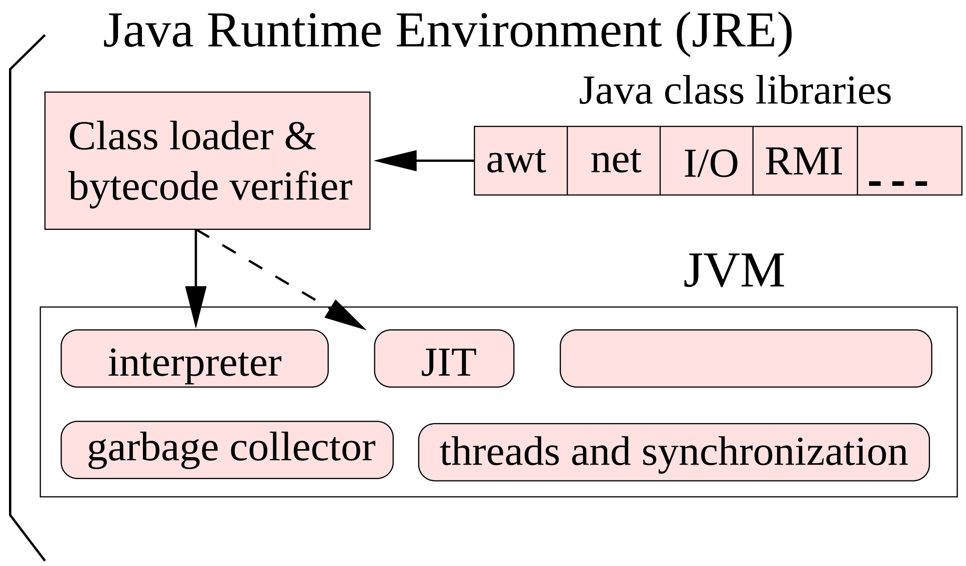

Figure 1.1: Java Runtime Environment

The Java API is a large set of standard services which help in the deployment of Java applications on different environments. The Java API provides objects and methods for threads, strings, graphics, user interface and other system functions. Although the implementation of such services will vary on different underlying systems, the Java specification requires that programs have access to a uniform set of API objects across platforms. The

Java API is designed to provide an abstract model for services. The Java API also include classes for directly accessing the underlying system. These classes include *native* methods for calling external (non-Java) libraries.

Java applications are compiled into machine-independent bytecodes. The bytecodes can be executed on different platforms having its own implementation of the JVM and a Java Base API. This makes Java an independent platform, since it can be run on any platform implementing the JVM and Java API.

1.1.1 Executing Java code

A Java application is compiled to the JVM instruction set (bytecodes) using a Java compiler. Java programs consist of different classes which are compiled into independent units called class files. Unlike traditional programs, the linking of these class files are done at run time. Classes can also be loaded dynamically at run time.

Bytecode can be viewed as object code for Java or its machine code. A hardware implementation of the JVM can execute bytecode directly, but other CPUs require an interpreter or compiler to translate the bytecodes into the CPUs native code as shown in the Figure 1.2.

An interpreter is a software emulation of the virtual machine. The interpreter translates bytecodes into native code of the processor. The interpreter is implemented using a loop that fetches, decodes and executes the bytecodes which constitute the program. The interpreter is easy to implement and does

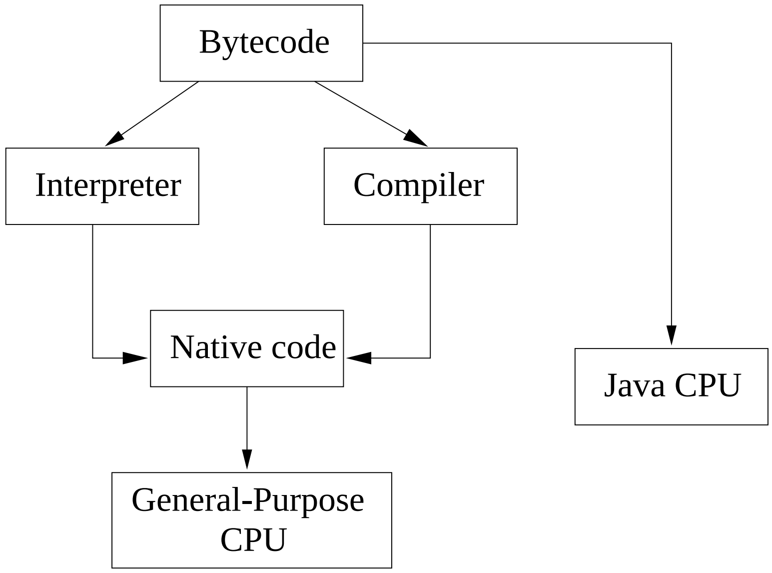

Figure 1.2: Three ways to execute Java bytecodes

not require a large amount of memory.

Another option to execute the bytecode is to compile it into the native code for the specific processor. To improve performance, Just-In-Time (JIT) compilers interact with the JVM at run time and compile appropriate bytecode sequences into native machine code. The JIT compiler performs certain optimizations when compiling the bytecodes to native code. Since the JIT compiler translates a sequence of bytecodes into native instructions, it can perform some simple optimizations. Some of the common optimizations performed by JIT compilers are data-flow analysis, translation from stack operations to register operations, reduction of memory accesses by register allocation, elimination of common sub-expressions etc. The higher the degree of optimization done by a JIT compiler, the more time it spends in the execution stage. Therefore a JIT compiler cannot afford to do all the optimizations that is done by a static compiler, both because of the overhead added to the execution time and because it only has a restricted view of the program.

A hardware implementation of the JVM (Java processor) will be able to execute the bytecodes without any software emulation layer. By directly executing the bytecodes, the overhead of the software emulation and the time spent in compilation can be eliminated. A hardware implementation of the JVM would have an instruction set which corresponds to the bytecode specifications and would include support for run time features like garbage collection, thread synchronization etc.

1.2 Problem Statement

All methods of Java execution involve some serious performance bottlenecks. The disadvantages associated with interpreters, JITs and hardware realizations of the JVM are outlined here. Offline compilers are not included since they do not comply with the complete Java specification.

1.2.1 Limitations of Interpreters

Interpreters are the most widely used approach to execute Java programs. A software interpreter emulates the JVM by fetching, decoding and executing bytecodes, and in the process maintains the contents of the computation stack and local memory state. Interpretation and other forms of software emulation of bytecodes are very slow relative to native compiled code execution of C or C++. Interpretation is typically slow because fetch and decode functionalities of normal program execution (reading and updating program counters, decoding the instruction, transferring control to activities that correspond to the opcode of the instruction just decoded, etc) are performed in software. While executing native compiled code the microprocessor hardware performs the fetch and decode actions. Therefore compiled code executes much faster compared to software translated code.

To improve Java execution optimized JVMs use JIT compilers [42, 11, 20] to compile Java methods (i.e. functions) instead of dynamically interpreting each bytecode. JIT compilers compile bytecodes into native code at run time, thereby adding compilation time to the total execution time. Initial JIT compilers were quick and dirty, generating unoptimized code for any method which was invoked. Current Java runtime environments are more sophisticated and mix interpretation and compilation using profiling information [30, 73]. This allows the compiler to spend more time on optimizing frequently executed methods, which lowers the execution time spent in the compiled methods. This method of execution which mix interpretation and JIT compilation is called adaptive or mixed-mode compilation.

1.2.2 Limitations of JIT compilers

JIT and adaptive compilers do not improve performance for all workloads. Programs which touch a lot of code and does not spend a majority of their execution time in loops can get better performance when executed using an interpreter. We use two micro-benchmarks to compare the execution times of different execution modes. Table 1.1 shows the execution cycles, instructions executed and IPC seen when the micro-benchmarks are executed using an interpreter, a JIT compiler and the HotSpot [30] adaptive compiler.

Execution	hello world			AddClass		
Mode	Cycles	Inst	IPC	Cycles	Inst	IPC
JDK 1.1.6 Interpreter	7.29M	21.76M	2.98	7.24M	21.64M	2.98
JDK 1.1.6 JIT	9.67M	27.85M	2.87	8.72M	25.50M	2.92
JDK 1.2 Interpreter	19.58M	46.44M	2.37	19.69M	46.95M	2.38
JDK 1.2 JIT	22.64M	55.80M	2.46	22.55M	55.79M	2.47
JDK 1.2.2 HotSpot	38.19M	88.23M	2.31	37.64M	88.45M	2.35

Table 1.1: Comparison of different software translation modes

The interpreter is seen to execute faster than the JIT and HotSpot compiler for both micro-benchmarks. The overhead of compiling a method is incurred at run-time and makes the JIT more expensive compared to the interpreter. Server applications which are typically long running applications and exhibit significant code reuse are ideal workloads for JIT compilers.

A breakdown of execution time shows that compilation constitutes a major fraction of total execution time in certain Java applications. Figure 1.3 shows the time spent during the execution of five benchmarks from the SPECjvm98 [7] benchmark suite, and the *hello world* micro-benchmark. It is observed that short running programs (*db* and *Hello World*) spend more time in translation than execution when run using JIT compilers. *Compress* is a long running benchmark which displays characteristics similar to server workloads. Not surprisingly, *compress* gains the most benefit from JIT execution, spending only a small fraction of the total execution time in translation.

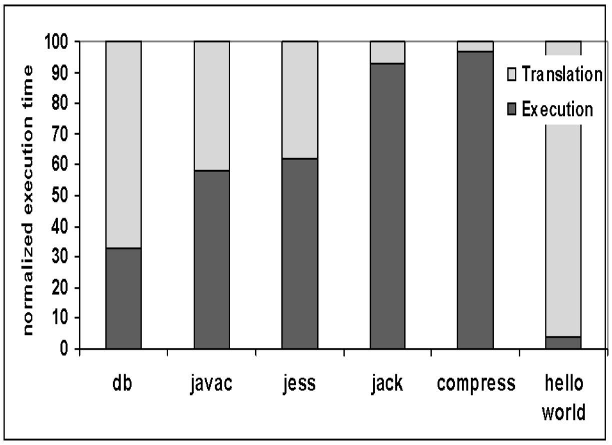

Figure 1.3: Breakdown of execution time

Another potential problem in JIT compilers is the increased number of data cache misses. Data cache misses during execution with a JIT compiler are dominated by compulsory write misses, caused by installation of the translated code [58]. Another observed phenomenon is the writing of translated code into the data cache and subsequent fetching of the same code from the instruction cache. This results in avoidable data transfer and double-caching [58]. Due to all these issues, JIT compilers are fruitful only in certain environments.

1.2.3 Limitations of Java Processors

Software emulation is easy to implement for new platforms but cannot offer a solution for fast execution of Java bytecodes. Interpreted execution of Java workloads requires, 35 SPARC instructions on the average to emulate each bytecode [62]. The average number of SPARC instructions required to emulate a bytecode is approximately 20 when using a JIT compiler [57]. These instruction counts include the interpretive and translative overhead of the interpreter and JIT respectively. The actual instructions required for the program to execute is only a small subset of the operations that happen during Java program execution.

Using hardware to execute the bytecodes directly eliminates the requirement of a software layer to emulate the bytecodes. Processors which implement the JVM in hardware and have bytecodes as their instruction sets are called Java processors. In modern high performance processors, much of the performance is obtained through instruction level parallelism (ILP). The stack nature of the JVM can significantly limit the bytecode parallelism that can be exploited in Java processors.

Figure 1.4 compares the available parallelism of Java programs to SPEC95 and C++ applications for a machine level parallelism (MLP) of 8, 16, 32, 64, 128 and infinite. Only true dependencies are considered while scheduling instructions, and perfect branch prediction and memory disambiguation is assumed. The latency of all operations is set to one cycle. The harmonic mean of the available parallelism for the various benchmarks are presented in Figure 1.4. Further details on the methodology used in this study can be found in [65]. The extremely low ILP of the interpreted Java programs can be attributed to the stack-based implementation of the Java Virtual Machine which imposes a strict ordering on the execution of the bytecodes. Limited ILP forms a critical performance bottleneck in stack-based Java processors, inhibiting high-performance.

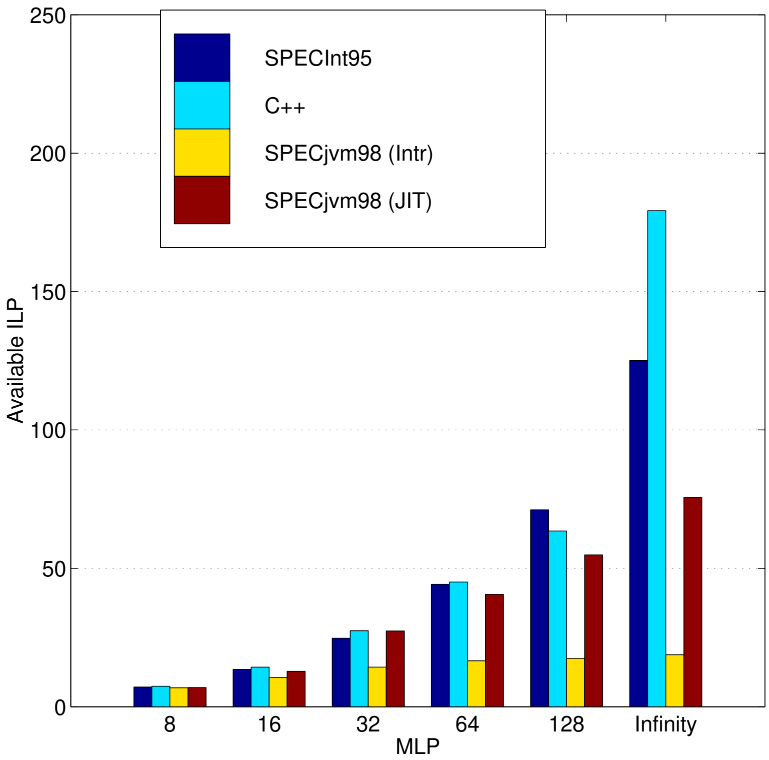

Figure 1.4: Available ILP in Java workloads

1.3 Proposed Approach

It is clear that each current execution mode for Java has its own strength and weaknesses. While software emulation is easy to implement, it incurs a high overhead during execution of certain Java workloads. A Java processor eliminates the software emulation layer but performance is limited due to limited available parallelism.

The execution of Java programs can be improved by a hardware accelerator or coprocessor that works in conjunction with a standard microprocessor. Essentially the effort is to bridge the semantic gap that exists between the bytecodes and the native instructions. The bytecode ISA is stack-based and uses few registers. General purpose RISC and CISC machines are register-based, and the *architectural gap*[1] between the JVM and these general purpose processors is large. If dedicated Java hardware such as the picoJava processor is used as a coprocessor, Java execution can be relegated to this Java processor and execution of other languages can be performed on the main processor. By using a dedicated Java processor, the emulating microarchitecture is brought to the level of the ISA being emulated, bridging the architectural gap. However, our ILP studies show that this is not the best way to bridge this gap, especially for high performance environments.

Another approach to improve Java execution is by providing hardware support in the processor to decode the bytecodes to native code. This is similar to the approach taken in the Pentium Pro processor [67] where complex X86 instructions are decoded dynamically in hardware to simpler instructions. The decoded instructions are executed in an aggressive RISC-style core optimized for executing fixed-length simple instructions. Original X86 instructions were complex and variable-length taking multiple cycles to execute. To alleviate this problem, the complex X86 instructions are decoded or translated into one or more fixed-length RISC instructions called micro-ops. Thus the RISC-style

[1] The architectural gap refers to the gap that exists between the emulated architecture (JVM) and the emulating architecture (the general purpose processor).

core is the emulating architecture and the original x86 ISA is the emulated ISA. In this dissertation we use a similar approach to decode the bytecodes to the native instructions of a general-purpose CPU. Hardware translation of Java bytecodes eliminate much of the high overhead associated with software translation and thus improves the performance of Java applications when executed on general purpose processors.

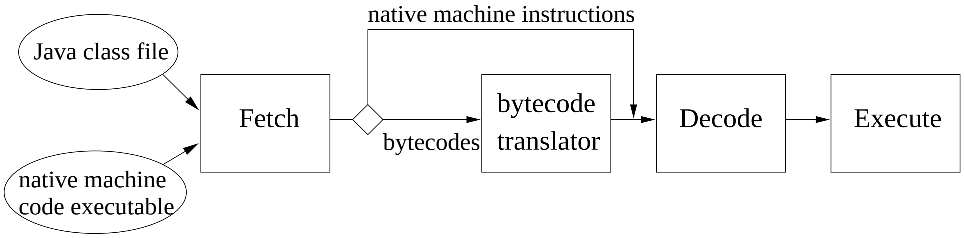

Figure 1.5: Hardware translation of Java bytecodes

Figure 1.5 shows a functional block diagram of the proposed architecture. A hardware translator is added to the general purpose processor to convert bytecodes to native instructions of the processor. If a non-bytecode (i.e. native) instruction is fetched, the hardware translator is bypassed and the instruction is sent directly to the processor core. This approach does not require any changes to the processor execution core or instruction set, and therefore does not degrade the performance of native code.

1.4 Goals of this dissertation

The focus of this dissertation is to provide microarchitectural support to enable efficient execution of Java programs on a wide range of platforms and execution environments. The goals of this dissertation are:

1. A comprehensive study on the execution characteristics of different Java execution modes using various benchmarks.

2. An understanding of the architecture implications of different JVM implementations to provide microarchitectural support for efficient Java runtime system development on different platforms and execution environments.

3. The design and evaluation of microarchitectural techniques to improve execution of Java programs.

In particular, the focus is on the design and implementation of hardware support for eliminating performance bottlenecks observed when executing Java on both general purpose processors, as well as a dedicated Java processor.

1. Hardware support for general-purpose processors: The most critical bottleneck observed during software emulation of Java applications is the translation of Java bytecodes to native instructions. Therefore, we investigate the performance improvements that can be obtained by providing hardware support for translation of bytecodes. Towards this, the dissertation focuses on:

   (a) Microarchitectural techniques that can be used to perform the translation of bytecodes to native instructions of the processor.

   (b) The sources of performance improvement when using a hardware translation over state of the art JITs and interpreters.

2. Hardware support for a Java processor: Java processors are stack machines which execute bytecodes directly in hardware. Such processors are used in the embedded market where system cost, power, area etc are issues given precedence over high performance. This dissertation concentrates on:

   (a) Investigation of cost-effective microarchitectural techniques to improve performance of the picoJava-II processor core, which is a widely used Java processor.

   (b) A detailed study of the verilog code of the picoJava-II core to propose microarchitectural techniques to exploit parallelism in the bytecode stream.

   (c) Removing false dependencies created by accesses to the Java stack which limit the exploitable ILP in the bytecode stream.

1.4.1 Outline of the Dissertation

Background and related work to this dissertation is presented in Chapter 2. We describe the benchmarks used for our research and perform a detailed characterization of different Java runtime systems in Chapter 3. The architectural impact of various execution modes and their limitations are studied, and we propose architectural enhancements to improve the performance of the Java runtime systems. In chapter 4, we propose the Hard-Int architecture to translate Java bytecodes to the native instruction set of a general purpose processor. Chapter 4 also describes the implementation of the Hard-Int architecture and presents a detailed performance evaluation, comparing the Hard-Int architecture to other Java execution modes. Chapter 5 presents microarchitectural techniques to exploit parallelism in an embedded Java processor. Experimental studies on the Verilog model of Sun Microsystems picoJava-II core are performed to propose cost-effective hardware support to improve performance. Concluding remarks are presented in Chapter 6.

Chapter 2

Background and Related Work

In this chapter we present some background on the Java execution environment and existing techniques to optimize Java execution. This chapter also provides a brief presentation of research efforts pertinent to this dissertation. The related work can be divided into different categories: workload characterization studies, efforts towards efficient Java runtime implementations and language specific hardware implementations.

2.1 Java Execution Environment

2.1.1 Java Virtual Machine

The JVM is the main component of the Java language specification. It is responsible for the cross-platform delivery, small size of the compiled code, security of the environment and for the management of local resources.

Figure 2.1 shows the high level components of a JVM. The JVM implements a stack architecture, which means that operations are carried using the stack as the source and destination for operands.

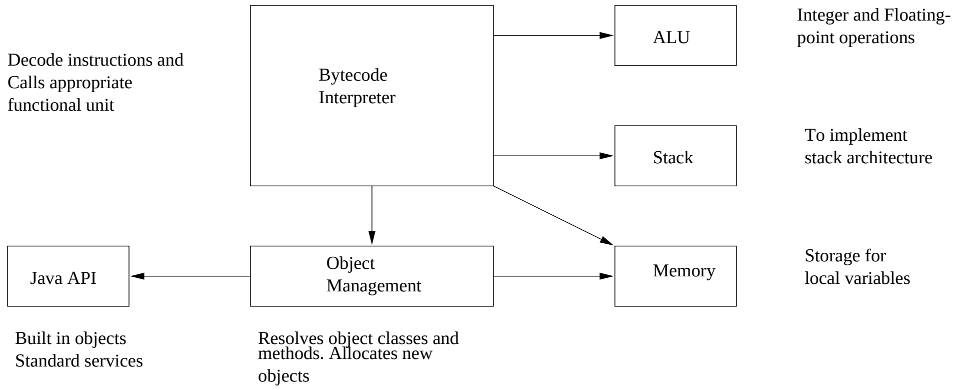

Figure 2.1: Components of the Java Virtual Machine.

In order to implement the JVM, it is necessary to provide all the functionality of a real computer and also adhere to the specifications of the Java language. The specification of the JVM takes care of maintaining a homogeneous run-time environment by providing a detailed description of the following components:

- Register Set
- Stack
- Execution Environment
- Constant Pool
- Object Management and Garbage Collection
- Method Area

- Basic Instruction Set

The registers of the JVM are similar to the registers in a processor, however, considering that the JVM is stack based, the registers are not used for passing or receiving arguments as in most machine registers. They hold the machine state and are updated for every line of bytecode executed.

The arguments to instructions must be pushed onto the stack before the instruction is executed, and an executing instruction places its result on the stack when it completes. The specification requires a traditional stack data structure that follows a *Last-in First-out* (LIFO) methodology and which expects the operands to be in a specific order.

The execution environment is maintained within the stack as a data set and is used to handle dynamic linking, normal method returns and exception generation. The execution environment handles dynamic linking by maintaining symbolic references to methods and variables for the current method and current class. A symbol table is used to translate these to actual calls.

The JVM maintains a special table for each class known as a *constant pool*. The constant pool contains string constants, class names, field names and other constant data objects that are referred to by the class structure or by the executing program. Because constants do not change, they are created at compile time. Items in the constant pool encode all names used by any method in a particular class. The information included in a class is the number of constants and the offset that specifies where a particular list of constants begins in the class description.

The global memory of a Java process is modeled as a collection of objects. When an object is created with a *new* instruction, a reference to that object is returned. This reference can be used in subsequent instructions, or stored in the current frame. An object persists in global memory until there are no references to it in any frame of the frame stack or in the constant pool of any visible object. When there are no such references, an object becomes *garbage*, and a special garbage collection process will reclaim its resources.

This *method area* is equivalent to the compiled code areas in the run-time environment used by other programming languages. It contains bytecode instructions that are associated with the methods in the compiled code and the symbol table needed for dynamic linking.

The Java compiler does not generate machine code in the sense of native hardware instructions. Rather, it generates bytecodes: a high-level, machine-independent code for the *hypothetical machine* implemented by the Java run-time system. The bytecodes used by Java resemble assembly instructions. Since these bytecode instructions represent low level operations, the programs are relatively fast compared to other interpreted languages [52].

Figure 2.2 illustrates the translation of Java source code using a Java compiler to a special class file format which consists of bytecodes. The class file consists of header information, sections which store constants information, class information, class field information and the methods themselves.

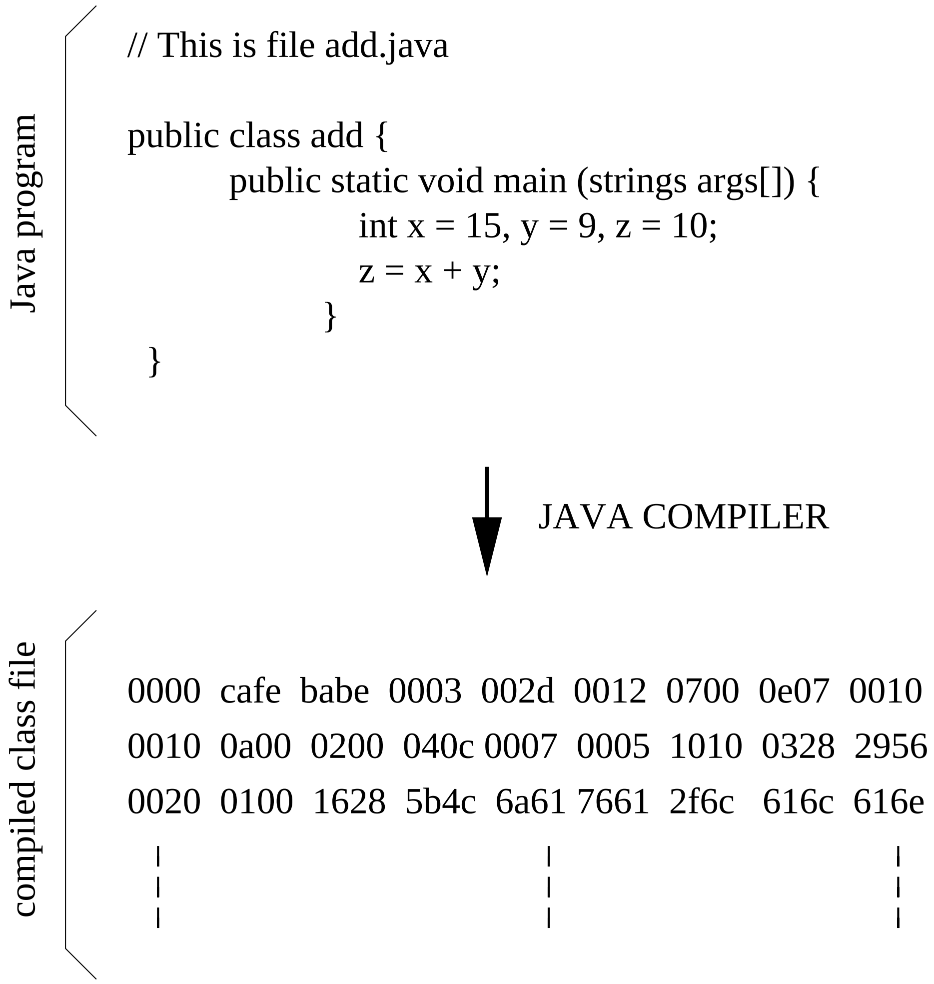

Figure 2.2: Translation of Java source code to bytecodes

2.1.2 Optimizing Java performance

The various alternatives to execute a Java application, presents researchers with different avenues for improving Java performance. We illustrate some of these techniques and provide examples for each of them.

1. **Better source compilers**: Java source-to-bytecode compilers are a potential area where classical compiler optimizations could lead to more optimized class files. Javac is the most commonly used Java compiler, which is developed by Sun Microsystems. Jikes [2] is a Java source to classfile compiler developed by IBM. It is important that Java compilers are fast, uses less memory and generates optimized code.

2. **Better JVMs**: The JVM can be made more efficient with better garbage collectors and thread monitors. Key techniques to achieving high performance and scalability in a JVM include fast responsive bytecode execution, and efficient and scalable synchronization. The ExactVM [4] forms the core of the JDK 1.2 Production Release for Solaris and achieves good performance. JVM performance can be improved by implementing high-performance garbage collection mechanisms and a fast memory system.

3. **Bytecode optimizers**: These tools recompile and optimize the class files and produce more efficient bytecode, which still adhere to the standard specification. These optimizers cannot make assumptions about the hardware, unlike a JIT compiler. DashO-Pro [22] is a bytecode opti-

mizer which is used for optimization, size reduction, and obfuscation of Java class files. The optimizer analyzes class files at the bytecode level to produce faster and smaller class files.

4. **Adaptive compilers**: These are dynamic compilers which profile a program while running and apply the appropriate optimizations. As the bytecodes are interpreted, a profiler keeps track of performance information and selects a method for compilation. Compiled methods are stored in a cache of native machine code. When a method is invoked, the compiled native code is used, if it exists in the cache else the method is interpreted. Sun's HotSpot execution engine [30] and IBM's mixed mode JIT compiler [73] are examples of adaptive compilers which are commonly used.

5. **Optimizing native compilers**: These are compilers which compile the bytecodes to native code or an intermediate language and can achieve near native performance. Since the compilation time using static compilers does not form a part of the application execution time, higher levels of optimizations can be performed. Vortex [39], NET [34] and Toba [54] are research compilers which translates bytecodes to native code. TowerJ [8] and GCJ [16] are commercial compilers that compile bytecodes to native code.

6. **Native method calls**: Calling native code for common library routines like graphics can lead to better performance. This will work as long as

the overhead in making the calls to native functions are low, and easy for programmers.

7. **Java CPUs**: A hardware implementation of the JVM will directly execute the bytecodes, removing the overhead associated with emulating it in software. The hardware implementation of the JVM can also provide run-time support for various functions like garbage collection and thread synchronization for higher performance. The picoJava [53] processor, PSC1000 from Patriot Scientific [5] and JEM [82] are examples of Java processors used in the embedded market.

2.2 Workload Characterization

This section briefly describes research performed in Java workload characterization. We also describe work done towards understanding the behavior of other object oriented languages, such as C++.

2.2.1 Object oriented workloads

Calder et. al. measured the behavior of a group of C and C++ programs and quantified the differences between them [17]. They found that programs written in the true object-oriented paradigm exhibited a greater use of both dynamic dispatch and dynamic calls than other C++ and C programs. C++ programs were seen to execute more procedure calls, indirect procedure calls and returns which implied that different branch prediction architectures are needed for C++ programs. The small size of procedures in C++ suggests the need for more inter-procedural analysis to take advantage of wide-issue architectures. A similar study was performed using hardware performance counters on the UltraSPARC-II processor. Factors including CPI, instruction and data cache misses, processor stalls due to instruction cache misses and branch misprediction, from real execution of several programs were measured and presented [59, 60].

In another study, Holzle and Ungar [33] look at the trade-offs between hardware and compiler optimizations for several large SELF applications. The need for specialized hardware for eliminating message dispatch overhead and tagged arithmetic in the SPARC architecture was seen to be less, when compiler optimizations were used. The hardware features that had a greater impact on performance was the instruction cache size and the data cache write policy. The performance impact of virtual functions and indirect branches was studied by Karel and Holzle [27, 26]. They found that the dispatch overhead of the standard virtual function table dispatch was significant. A potential for improvement in indirect branch prediction with non-hybrid predictors was observed.

Aigner and Holzle discuss the impact of two object-oriented specific optimizations, type feedback and static optimizations for C++ [12]. Type feedback is a profile based optimization technique which uses profile information at run time to eliminate dynamic dispatches. A static optimization technique like class hierarchy analysis, can statically bind some virtual function calls given the applications complete class hierarchy.

In this dissertation we do not look at optimizations that are targeted specifically at alleviating the bottlenecks caused by the object-oriented features of the Java language. Our workload characterization demonstrated that bytecode translation is the most critical overhead during Java execution and hence we focus on translation issues. Object-oriented optimizations are orthogonal to the solution presented in this dissertation for improving Java performance.

2.2.2 Java workloads

Previous research has mainly concentrated on analyzing the interpreted behavior of Java execution. Romer et.al. [74] investigated the interaction of interpreters with modern architectures and demonstrated that interpreter performance is relatively independent of the application. They also concluded that specialized hardware support for interpreted environments was not essential and that performance improvements could be achieved through software means. We concur to them in that highly specialized processors are not required for Java. However we favor adding hardware assists for specific tasks such as translation which incur a high overhead when performed in software. A 2-issue DEC Alpha 21064 was used by Romer et. al. for their study, whereas we used a 4 and 16 way superscalar machine. As the width of the processor increases the penalty of branch mispredictions and cache misses increase drastically. An overhead of 16 instructions per bytecode was measured by Romer et. al. to perform fetch and decode actions. We also observe a high overhead for the translation process during interpretation and JIT compilation. Although they suggest that software techniques can be used to alleviate the overhead, we favor hardware support for handling this issue. We still maintain the position that general purpose processors with minor hardware enhancements can yield better Java performance.1

Hsieh, et.al. [35] studied the impact of interpreters and offline Java compilers on microarchitectural resources such as the cache and the branch predictor. They attribute the inefficient use of the microarchitectural resources by the interpreter as a significant performance penalty. It is also observed in their work that an offline bytecode to native code translator is a more efficient Java execution model for utilizing the caches and branch predictors. However, JIT compilers were omitted in Hsieh's work [35]. We perform a similar study using JIT compilers and observe that JITs make poor utilization of the cache and show locality worse than interpreters. we identify the reasons for the poor locality exhibited during JIT compilation. Several tradeoffs concerning the design of caches, branch predictors, and exploitation of parallelism using ILP techniques are explored in this dissertation.

Characterization of Java at the bytecode level is important for implementation of efficient Java run time systems [10, 62]. An analysis at bytecode level points to processor features required in a Java processor or in a software JVM. The bytecode analysis also helps to explain many observations at the machine code level, generated using an interpreter or JIT compiler. Workload studies characterizing Java at the bytecode level show that bytecodes have a high frequency of memory references and show good instruction locality [62].

2.3 Software Optimizations

In this section we describe research done in optimizing Java performance using software techniques. Currently, a significant amount of effort is spent towards building sophisticated JIT compilers and providing more efficient runtime support for Java execution.

Since the earlier JVM implementations were interpreter-based their performance was disappointing. A number of software solutions and hardware solutions were proposed to improve the performance. Numerous JIT compilers became available, such as Kaffe [80], Cacao [42] and Guava [31]. The optimizations used in the Sun JIT compilers are explained in [38]. The approach proposed in this dissertation is a first-cut at using hardware to perform bytecode translation and does not include such optimizations as performed in a JIT compiler. However, our proposal can be viewed as a first pass of the JIT compiler and additional hardware can be used to perform more sophisticated optimizations on the translated code. JIT compilers also exist for other platforms such as the x86 [11] and Alpha [42]. Adaptive or dynamic compilers such as HotSpot [30] are more efficient than JITs since they selectively compile methods based on profiling the program execution.

Offline compilers such as NET [34] and Vortex [39] compile the bytecodes to native code. The translation scheme of Caffeine system which also compiles bytecode to native code is described in [36]. A mapping of the JVM stack model to a register computation model performed during translation by Caffeine was observed to result in significant performance gain over an interpreter. Other offline compilers like Toba [54] and Harissa [51] converts the bytecodes to an intermediate language like C, which can be optimized by a C compiler to generate efficient native code. Bytecode optimizers such as DashO [22] analyze the class files and produce faster and more compact class files.

DAISY(Dynamically Architected Instruction Set from Yorktown) is a VLIW architecture developed at IBM for fast execution of x86, PowerPC and Java code [13]. A JIT compilation technique is used to achieve compatibility with different architectures. DAISY introduces simple hardware features intended to simplify emulation of existing architectures, so that all existing software for an existing architecture runs without changes on the VLIW architecture. DAISY performs dynamic translation and parallelization across multiple paths and loop iteration boundaries to execute code on the VLIW architecture. A VLIW Java processor based on JIT compilation was proposed as an extension of DAISY [40]. We feel that using binary translation will show performance improvement for long running server applications. However Java is also used for developing client applications and using JIT translation for these applications will cause a performance degradation compared to interpreters, or the solution proposed in this dissertation.

The technology used in Transmeta's Crusoe processors [41] is also very similar to the approach taken in DAISY [13]. The technology essentially consists of a VLIW hardware engine logically surrounded by a software layer.

Code Morphing software is used to translate X86 native instructions to the Crusoe native instructions. Along the lines of a JIT compiler, a simple interpretation is performed initially and the translated code is stored in memory. On subsequent execution of the translated code, optimizations are performed to make it execute faster. The Crusoe hardware is tailored to support binary translation or Code Morphing by implementing techniques such as shadow registers, alias hardware etc. The proposal in this dissertation is to use additional hardware to perform bytecode translation, which is in contrast to the philosophy adopted by Transmeta. Based on available literature, the Code Morphing software approach is similar to JIT compilers, and we show that hardware interpretation is more efficient than JIT compilation.

Runtime support in the form of synchronization mechanisms [15], as well as class loading and resolution can play a crucial role in execution/interpretation overheads (they are common to an interpreter as well as a JIT-based execution). There have been attempts to improve the performance of the synchronization operations using cooperative thread scheduling [43], or by making special provision for faster execution of single-threaded user programs [51, 54].

2.4 Language specific hardware

In this section we briefly describe previous research in building language specific hardware, or the design of hardware for efficient execution of a particular language. We conclude with a description of research done towards building dedicated hardware for executing Java bytecodes.

There have been various attempts to build specialized hardware to support specific language environments in the past. The idea of Virtual Machines, which was made ubiquitous by the popularity of Java, dates back to the 1960s. UNCOL (UNiversal COmputer Language) [70] is an intermediate language, which was proposed in 1958 to reduce the developmental effort of compiling many different languages to different architectures. P code [64] is a virtual machine used by different Pascal implementations popular in the late 1970s. Like today's Java, it was based on a virtual machine with a standard set of low-level, machine-language-like p-code instructions that were emulated on different hardware. A Pascal compiler that emitted p-code executables could produce a program that could be run under the P-System on different platforms.

SOAR (Smalltalk On A RISC) [21] modified the RISC II design to support Smalltalk. The SOAR architecture provides support for frequent method invocations, by implementing overlapping of register windows. It also supports inline caching to find the targets of virtual call sites and tag checks for helping generational garbage collection [75]. The Mushroom project provides hardware support for object memory management, garbage collection, dynamic binding and extensible instruction sets [81] [37]. Language based architectures have also been proposed for execution of languages like FORTH, Lisp and Prolog. FORTH chips are used in the embedded market and are essentially stack based architectures. Symbolics, Inc was a producer of computer systems for running and developing object-oriented programs in Lisp. It designed and built workstations as well as writing an operating system and development environment to run on those workstations.

Our work differs from the aforementioned research on language specific processors in that we are targeting a general purpose system that runs programs in all languages rather than Java alone. The language specific processors listed above cater only for programs in their specific languages. Our primary goal is to improve Java in the general purpose processor domain.

2.4.1 Java processors

Java processors can be classified based on the amount of support that is provided for the direct execution of Java bytecodes. PicoJava [53] and JEM1 [82] implement bytecodes as the native code, whereas the ShBoom processor has an instruction set similar to that of the JVM [5].

The picoJava architecture is a microprocessor core to directly execute the Java bytecode instructions and provide hardware support for other essential functions of the JVM [53]. It uses a stack cache to eliminate inefficiencies associated with stack-based instruction processing and hardware support for garbage collection. The bytecode instructions are classified according to their complexity and implemented using hardware, microcode and software emulation based on their category. A register file is used to cache the top 64 entries of the stack, for efficient processing. Optimizations such as *instruction folding*, tries to solve the problem of access inefficiency in a stack machine. picoJava also provides hardware support for run time functions of the JVM like thread management and object handling. Since picoJava is targeted for set top boxes and low cost network terminals, it was designed keeping the die area, cost and power consumption in mind, as opposed to high performance.

Different studies have investigated the performance benefits of various architectural support features for a Java processor [76, 56]. The impact of adding support for stack processing, object manipulation and virtual method instructions was studied by Vijaykrishnan [76]. An extended folding scheme to eliminate operations that moved data between the local variables of the stack frame and the operand stack was proposed, and was seen to eliminate redundant data movement operations. An object cache addressed virtually by the object reference and field offset was seen to eliminate the overhead associated with accessing objects using the picoJava's handle representation. A path history based predictor was seen to improve the prediction rate of indirect branches as compared to a branch target buffer scheme. Hybrid cache schemes were proposed, which provided efficient virtual method invocation. Our workload characterization showed that translation is the most critical bottleneck and hence this dissertation concentrates on alleviating the translation overhead. Adding support for object manipulation and virtual method invocations may further help to improve the performance of Java workloads on the proposed architecture, however this is yet to be investigated.

The MAJC (Microprocessor Architecture for Java) chip [48] proposed by Sun Microsystems uses thread level parallelism and speculation to improve performance of Java applications. Java methods are mapped to threads and executed speculatively to reduce execution time. A JIT compiler is used to perform the translation of bytecodes to the native code of MAJC. The MAJC processor tries to achieve high Java performance by parallel execution of the Java methods, but still incurs the translation overhead. The architecture proposed in this dissertation attacks the translation bottleneck, and certainly, it can be extended to speculatively execute different bytecode streams.

The JSTAR accelerator [68] recently announced by JEDI Technologies is a Java coprocessor for general purpose embedded processors. JSTAR is a coprocessor that interfaces to the native microprocessor core and its cache or memory subsystem. JSTAR fetches bytecodes from memory and executes them in conjunction with the native processor. A speedup of 5.5 over an interpreter was achieved when on a MIPS R3000 processor core [68]. While available literature mentions that JSTAR performs translation using hardware, details on the architecture or implementation have not been revealed.

Chapter 3

Characterization of Java Runtime Systems

Interpretation, JIT compilation, and hardware realization are well known solutions for a JVM, and previous research has proposed optimizations for each of these techniques. However, each technique has its pros and cons and may not be uniformly attractive for all hardware platforms. Instead, an understanding of the architectural implications of JVM implementations with real applications, can be crucial to the development of enabling technologies for efficient Java runtime system development on a wide range of platforms. Towards this goal, this chapter examines architectural issues, from both the hardware and JVM implementation perspectives.

We start with an investigation of the CPU and cache architectural support that would benefit JVM implementations. Cache and branch predictor performances are investigated in detail. An investigation of the available parallelism during the different execution modes, using applications from the SPECjvm98 benchmarks is also performed.

3.1 The SPECjvm98 Benchmark Suite

The SPECjvm98 benchmark suite [7] is used to compare the different Java execution modes, and to evaluate the proposed techniques in this dissertation. The SPECjvm98 benchmark suite contains eight different tests, five of which are either real applications or derived from real applications that are commercially available. SPECjvm98 allows users to evaluate performance for the hardware and software aspects of the JVM platform. On the software side, it measures the efficiency of JVM, the JIT compiler, and operating system implementations. On the hardware side, it includes CPU, cache, memory, and other platform-specific features.

Benchmark	Description
compress	A popular LZW compression program.
jess	NASA's popular CLIPS rule-based expert systems.
db	Data management benchmarking software written by IBM.
javac	The JDK Java compiler from Sun Microsystems.
mpegaudio	Software algorithm to decode an MPEG-3 audio stream.
mtrt	A dual-threaded program that ray traces an image file.
jack	A real parser-generator from Sun Microsystems.

Table 3.1: Description of the SPECjvm98 Benchmarks

A summary of the benchmarks is provided in Table 3.1. The benchmarks can be run using three data sets (-s1, -s10 and -s100 are the arguments used to specify the data set to be used when running the benchmarks). All our benchmarks were run at the command line prompt (using the -s1 data set), as opposed to being run from an applet in order to capture the the program characteristics. Table 3.2 shows the dynamic instruction count for the SPECjvm98 benchmarks when run using the three data sets.

		Dynamic Instruction Count		
Benchmark	Exec. mode	**-s1**	**-s10**	**-s100**
compress	Interpreter	10425544771	12266399319	135574216217
	JIT	1385557173	1630087506	11908766544
jess	Interpreter	259702575	1883530288	37278365517
	JIT	188581319	597550362	16917095279
db	Interpreter	86844476	2563156018	71418597792
	JIT	75959798	1954427130	39318582607
javac	Interpreter	199254389	1686869000	44062969591
	JIT	167556898	1057087495	23210316813
mpeg	Interpreter	1314397268	13169917270	123755136190
	JIT	264879641	2049987114	18407753666
mtrt	Interpreter	1531909956	4672038088	46691888750
	JIT	942593921	1853119781	13830116339
jack	Interpreter	2668899901	5254167683	43832392904
	JIT	986682716	1839251078	14851289939

Table 3.2: Dynamic instruction count for all the three data sets

3.1.1 Method Reuse

We use a metric called *method reuse* factor to understand each Java application. The *method reuse* factor can be defined as the ratio of *method calls* to *number of methods*, which is presented in Table 3.3. The performance benefits that can be obtained from using a JIT compiler is directly proportional to the *method reuse* factor, since the cost of compilation is amortized over multiple calls in JIT execution. Table 3.3 also shows that the static size of the benchmarks remain constant across the different data sets (since the number of unique methods does not vary), although the dynamic instruction count increases for the bigger data sets (due to increased method calls).

	s1		**s10**		**s100**	
benchmarks	calls	methods	calls	methods	calls	methods
compress	17330744	572	18170275	570	14566857	572
db	65379	642	1610941	645	91753107	658
jack	2318110	1230	4621508	1233	39172145	1240
mpegaudio	954605	843	8289656	846	93046042	844
jess	414349	1222	5697628	1313	95957670	1375
javac	213243	1384	2515940	3142	54503910	3325
mtrt	1906112	781	7031487	785	71168982	796

Table 3.3: Total number of method calls (dynamic) and unique methods for the three data sets

Looking at the three data sets, we see that there is very little difference in the number of methods across data sets. The higher number of method calls indicate that the method reuse in the benchmarks for higher data sets would be substantially more. This would then lead to better performance for the JITs, as observed in Table 3.2. Table 3.2 shows that the instruction count when the benchmarks are executed using a JIT is much lower than using an interpreter for the s100 data set. Since there is higher method reuse in all benchmarks for the larger data sets, using a JIT results in better performance over an interpreter.

3.1.2 Instructions Executed per Bytecode

Data presented in Table 3.4 and shows how many SPARC instructions on the average are needed in each benchmark to execute a bytecode. This will represent the cost of interpretation or execution cost using a JIT compiler for the benchmarks.

benchmarks	bytecode count	Instructions per bytecode	
		interpreter	JIT
compress	954990234	10.91	1.45
jess	8126332	31.93	23.20
db	2035798	42.64	37.31
javac	5958654	33.37	28.12
mpeg	115748387	11.35	2.28
mtrt	50683565	30.22	18.59
jack	175740325	15.18	5.61

Table 3.4: Number of instructions executed per bytecode

It is observed that the instructions executed per bytecode go down to as low as 1.45 for *compress*. Benchmarks *compress* and *mpeg* had the least cost for the interpreter mode, and for the JIT mode of execution it goes down further. This shows that using a JIT removes almost all overhead in terms of instructions generated. For the other benchmarks the emulation cost higher, varying from 5.6 for *jack* to 42.6 for *db*.

3.2 Architectural Issues

Understanding the underlying characteristics of Java applications in their various modes of execution, and in comparison to code in other languages/paradigms is extremely important to develop an efficient run-time environment for Java. Two popular JVM implementations have been used in this study: the Sun JDK 1.1.6 [4] and Kaffe VM 0.9.2 [3]. Both these JVM implementations support the

JIT and interpreted mode. We conduct detailed studies on the instruction mix of SPECjvm98 programs in interpreter and JIT-compiled modes of execution. The cache performance, branch predictor performance, and the instruction level parallelism of these programs, at the native SPARC code level is studied in detail. To capture architectural interactions, we have obtained traces using the Shade binary instrumentation tool [19] while running the benchmarks under different execution modes. Our cache simulations use the *cachesim5* simulators available in the *Shade* suite, while branch predictors have been developed in-house. The instruction level parallelism studies are performed utilizing a cycle-accurate superscalar processor simulator. This simulator, can be configured to a variety of out-of-order multiple issue configurations with desired cache and branch predictors.

3.2.1 Instruction Mix

Figure 3.1 shows a summary of the results on the instruction mix, computed cumulatively over all the SPECjvm98 programs. The individual application mixes exhibit a similar trend, and the individual instruction mix for each benchmark is presented in Table 3.5.

Execution in the Java paradigm, either using the interpreter or JIT compiler, results in 15% to 20% control transfer instructions and 25% to 40% memory access instructions, a behavior not significantly different from traditional C and C++ programs [17]. Although memory accesses are observed to be frequent in the instruction stream in both modes, it is more frequent (by

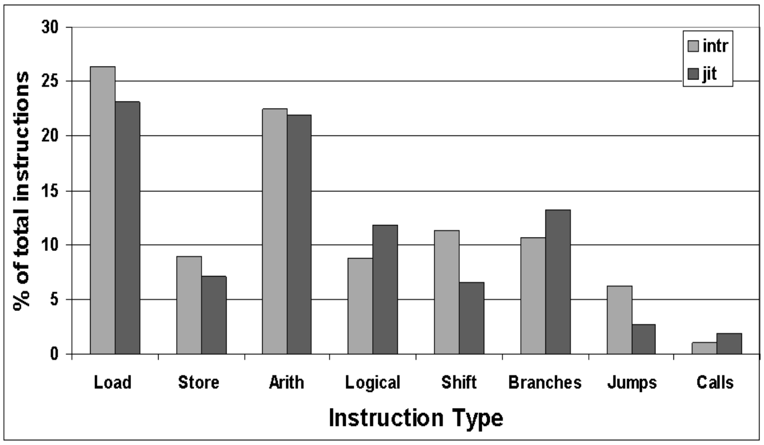

Figure 3.1: Instruction mix for the SPECjvm98 benchmarks

Benchmark		Control Transfer			Computational Inst			Memory	
		Branch	Call	Jump	Arith	Logical	Shift	Load	Store
compress	Intr.	5.57	0.01	9.17	20.15	5.48	14.95	33.35	10.56
	JIT	5.61	0.23	2.50	13.49	10.31	7.71	40.61	6.77
jess	Intr.	12.85	1.72	4.96	22.94	10.27	9.04	22.93	8.34
	JIT	15.21	2.87	3.44	23.78	13.05	5.34	16.61	7.59
db	Intr.	13.91	2.00	4.50	24.60	11.14	8.13	20.29	7.65
	JIT	15.73	2.73	3.14	24.98	13.19	5.54	16.04	7.61
javac	Intr.	13.35	1.69	4.71	24.55	10.32	8.96	21.93	7.89
	JIT	15.90	2.50	2.91	25.52	13.13	5.62	16.55	7.35
mpeg	Intr.	5.54	0.13	8.94	21.55	4.97	15.74	31.80	9.65
	JIT	6.91	0.85	1.45	17.37	8.52	9.96	31.89	6.49
mtrt	Intr.	14.04	0.67	3.96	22.54	11.50	9.38	25.18	8.85
	JIT	17.81	1.31	2.41	26.28	14.06	5.76	18.31	6.17
jack	Intr.	9.36	0.70	7.31	20.64	7.80	12.57	29.10	9.62
	JIT	14.69	2.31	2.88	21.65	10.48	5.47	22.31	7.96

Table 3.5: Instruction Usage Characteristics for the SPEC JVM98

This table shows the instruction characteristics for the different benchmarks. All the numbers are obtained by running the benchmarks using -s1 option.

approximately 5%) in the interpreted mode in comparison to the JIT compiler. In interpreted mode, a large percentage of operations involve accessing the stack, which translate to loads and stores in the native code. Contrary to this, in the JIT compiler mode, many of these stack operations are optimized to register operations, resulting in a reduction in the frequency of memory operations in the instruction mix. While this is to be expected, our experiments quantify the percentage reduction.

Past studies have shown that Java applications (and object oriented programs in general) contain indirect branches with a higher frequency than SPECint programs [28, 27]. Information on indirect branch (or jump) frequency in interpreted and JIT execution modes are provided in Table 3.5. Comparing the two execution modes, the interpreter mode has higher percentage of jumps (primarily due to register indirect jumps used to implement the switch statement in the interpreter, and due to high frequency of virtual function calls), while the code in the JIT compiler case has higher percentage of branches and calls. JIT compilers optimize virtual function calls by inlining those calls, thereby lowering the number of indirect jump instructions. A combination of a lower number of switch statements that are executed and inlining of the method calls, results in more predictable behavior of branches in the JIT mode, as illustrated in the next section.

3.2.2 Branch Prediction

The predictability of branches in Java applications along with the suitability of traditional branch predictors, is examined in this section. Indirect jumps and virtual function calls that are abundant in Java applications, especially in the interpreted mode of execution, can complicate the task of predicting the outcome of these control instructions. Table 3.6 illustrates the branch misprediction rates for four different branch prediction schemes including a simple 2-bit predictor [32], 1 level Branch History table (BHT) [32], Gshare [49] and a two-level predictor indexed by PC (described as GAp by Yeh and Patt [83]).

Benchmark	Exec. mode	**2-BIT**	**BHT**	**Gshare**	**GAp**
compress	interpreter	60.23	34.69	34.90	35.41
	JIT	28.31	9.26	8.97	8.93
jess	interpreter	46.39	19.13	19.37	18.66
	JIT	38.16	13.05	12.74	12.88
db	interpreter	43.97	17.12	16.82	16.69
	JIT	39.64	12.82	12.70	12.81
javac	interpreter	44.74	18.04	17.90	17.39
	JIT	39.06	12.92	12.18	12.47
mpegaudio	interpreter	52.54	31.61	33.29	31.59
	JIT	37.47	12.24	11.88	12.16
mtrt	interpreter	48.78	14.54	13.17	13.42
	JIT	41.43	11.62	9.20	10.44
jack	interpreter	56.26	28.31	28.78	27.92
	JIT	35.68	12.78	12.26	12.65

Table 3.6: Branch misprediction rates for four predictors

The one level predictor and GShare predictor have 2K entries and the two level predictor has 256 rows of 8 counters each (3 bits of correlation). The Gshare predictor uses 5 bits of global history. The Branch Target Buffer (BTB) contains 1K entries. The branch predictors get sophisticated as we go from left to right in Table 3.6. The simple 2-bit predictor has been included only for validation and consistency checking. As expected from trends in previous research, among the predictors studied, Gshare or GAp has the best performance for the different programs. The major trend observed from our experiments is that the branch prediction accuracy in interpreter mode is significantly worse than that for the JIT compiler mode. This is a direct implication of the control transfer instruction mix in the interpreter and JIT compile modes. The interpreter mode results in a high frequency of indirect control transfers due to indirect jumps used to implement virtual method calls and the switch statement for case by case interpretation. The accuracy of prediction for the Gshare scheme is only 65 to 87% in interpreter mode and 88 to 92% in the JIT compiler mode.

Thus, it is observed that branch predictor performance for Java applications is significantly deteriorated by the indirect branches abundant in the interpreter mode, whereas execution with the JIT compiler results in performance comparable to that of traditional programs. Hence we conclude that if Java applications are run using the JIT compiler, the default branch predictor would deliver reasonable performance, whereas if the interpreter mode is used, a predictor well-tailored for indirect branches (such as [27], [77]) should be used.

3.2.3 Locality and Cache Performance

In addition to examining the locality/cache behavior of Java executions in the following discussion, we also examine how the coexistence of the JVM and the application being executed affects the locality behavior of the entire execution. A detailed study of the cache behavior is conducted, looking at the entire execution in totality, as well as the translation and execution parts (of the JIT mode) in isolation.

Benchmark		**I-Cache**			**D-Cache**		
		Refs	Misses	% miss	Refs	Misses	%miss
compress	Intr.	10425M	84398	0.001	5365M	21M	0.394
	JIT	1385M	218101	0.013	751M	43M	5.828
jess	Intr.	259M	179545	0.068	81M	2.3M	2.910
	JIT	188M	616962	0.321	45M	3.7M	8.298
db	Intr.	86M	82873	0.094	24M	751592	3.097
	JIT	75M	232046	0.300	17M	1.2M	6.764
javac	Intr.	199M	140239	0.069	59M	1.8M	3.099
	JIT	167M	469143	0.274	40M	3.3M	8.335
mpeg	Intr.	1314M	92439	0.007	544M	1.9M	0.356
	JIT	264M	355896	0.134	101M	3.2M	3.232
mtrt	Intr.	1531M	252370	0.016	521M	8.6M	1.654
	JIT	942M	522692	0.054	230M	16M	7.239
jack	Intr.	2668M	124563	0.005	1033M	11M	1.069
	JIT	986M	1.0M	0.102	298M	15M	5.341

Table 3.7: Cache Performance for the SPECjvm98 benchmarks

This table shows the number of references, misses and miss rate for the instruction and data cache. Cache size= 64K bytes, block size= 32 bytes, I-cache is 2-way and D-cache is 4-way set-associative. M - indicates million.

Table 3.7 illustrates the number of references and misses for the L1 instruction and data cache in the interpreter and JIT compiled modes. Both instruction and data caches are of 64K bytes size and have a block size of 32 bytes. The instruction cache is 2-way set associative and the data cache is 4-way set associative. Instruction cache performance in interpreted mode is extremely good with hit-rates higher than 99.9% in all benchmarks. The interpreter is a switch statement with approximately 220 cases for decoding each bytecode. The excellent instruction locality in interpreted mode stems from the fact that the entire switch statement or at least the most frequently used parts of it nicely fit into state-of-the-art cache sizes. Locality studies of Java bytecodes [62] illustrates that less than 20% of distinct bytecodes account for 90% of the dynamic bytecode stream in all the programs. In fact, the aforementioned study showed that 15 unique bytecodes accounted for 60% to 85% of the dynamic bytecode stream of the SPECjvm98 programs, and 22 to 48 distinct bytecodes constituted 90% of the bytecode stream. These factors result in a small working set for the interpreter.

The instruction cache performance in JIT compiler mode is inferior to instruction cache performance in interpreter mode. Dynamically compiled code for consecutively called methods may not be located in contiguous locations. Rather than bytecode locality, it is method locality, method footprint, and working set properties of the JIT compiler code that determine the instruction cache performance for the execution of code generated in the JIT mode. Compilers typically result in poor cache performance (as exemplified by *gcc* in the SPEC suite [17]) and compilation process is a major component of the JIT mode. For applications like *db*, *jess* and *javac* which spend a significant amount of time in the translation part (see Figure 1.3), the I-cache misses are more dominant.

The data cache performance of Java applications is worse than its instruction cache performance, as is the case for normal C/C++ programs. However, data locality in the interpreted mode is better than the locality in the case of the JIT compiler. In the interpreter mode, each time a method is executed, the bytecodes are accessed from the data cache and decoded by the interpreter. The intended code is thus treated as data by the interpreter, in addition to the actual data accessed by the application, resulting in a lower miss rate overall (code usually has better locality than data). The benchmark data and benchmark bytecodes will be allocated and accessed from the data cache.

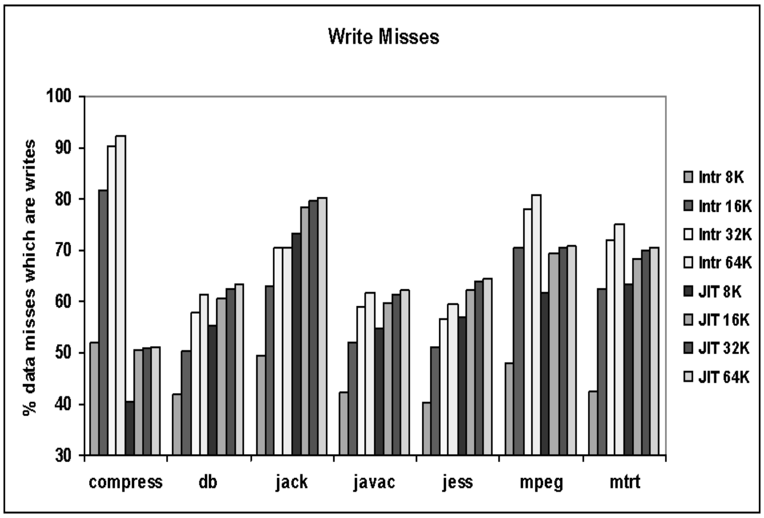

Figure 3.2: Percentage of Data Misses that are Writes. Cache used is direct mapped with line size of 32 bytes

Two benchmarks, *compress* and *mpeg*, exhibit significant method reuse and yield excellent data cache hit ratios in the interpreter mode, because the footprint can be entirely captured in the cache. In contrast, the JIT compiler translates the bytecodes fetched from the data cache into native code before the first execution of the method. Therefore the subsequent invocations of the method do not access the data cache (they access the I-cache) for bytecodes. This results in a drastic reduction of total data cache references from interpreter mode to JIT mode as illustrated in Table 3.7. The number of data references in the JIT compiler case is only 20% to 80% of the reference count in the interpreter case. Of the total data cache misses in the JIT mode, 50 to 90% of misses at 64K cache size are write misses (as illustrated in Figure 3.2).

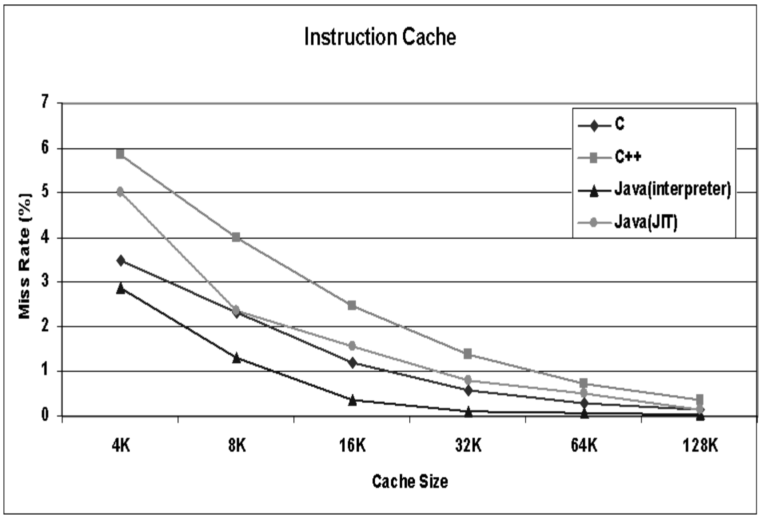

Figure 3.3: Instruction Cache Miss rates for C, C++ and Java workloads. The miss rates for C and C++ are obtained from studies by Calder et. al

Figures 3.3 and 3.4 illustrate the average instruction and data cache miss rates of SPECjvm98 workloads in comparison to the SPECint programs and several C++ programs. For both instruction and data caches, the interpreter mode exhibits better hit rates than C, C++ and the JIT mode of execution. The behavior during execution with the JIT compiler is closer to

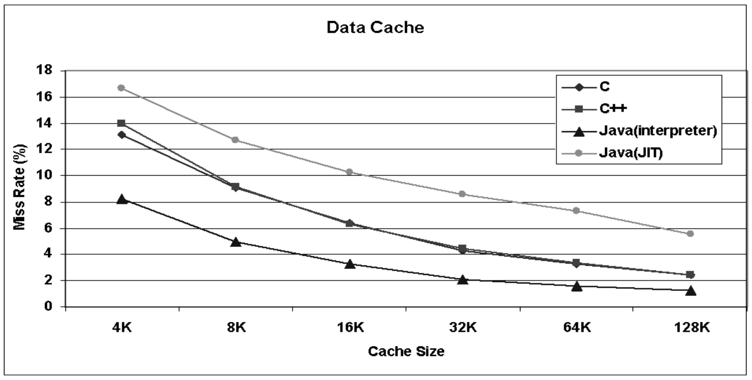

Figure 3.4: Data Cache Miss rates for C, C++ and Java workloads. The miss rates for C and C++ are obtained from studies by Calder et. al

that of traditional C and C++ programs for the instruction cache. In the case of data cache, the miss rates for the JIT mode of execution are highest among the different workloads. It may be inferred that the behavior of Java applications are predominantly dependent on the execution mode rather than the object-oriented nature of the language i.e. the results depend more on whether they are run in interpreter or JIT mode rather than on the fact that they are object-oriented.

One noticeable fact in Table 3.7 is that the absolute number of misses (instruction and data) in the JIT compiler mode is higher than the number of misses in the interpreter mode, despite the reduction in total instruction count and data cache reference count. There are two factors that can be attributed to this - code generation and installation of translated code performed by the JIT compiler. Both these operations can result in a significant number of misses, which we show by studying the behavior of these references in isolation.

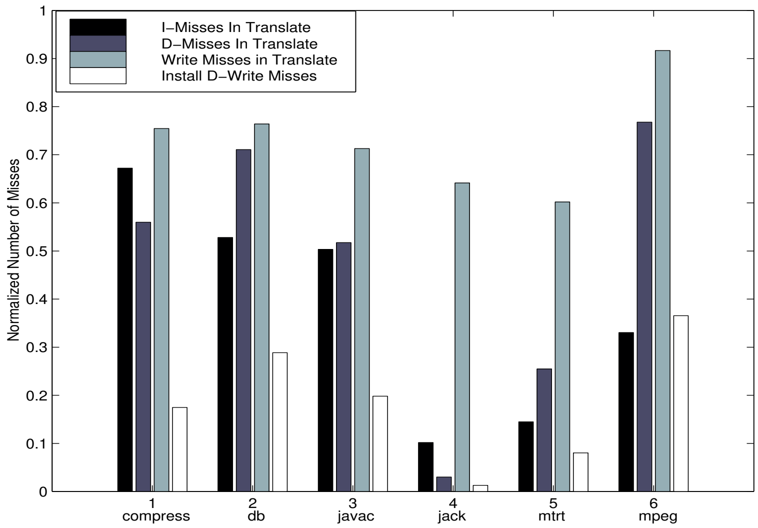

Figure 3.5: Cache Misses within Translate Portion. Cache configuration used : 4-way set associative, 64K DCache with a line size of 32 bytes and 2-way set associative, 64K ICache with a line size of 32 bytes.

We have further isolated the cache behavior during the translation part and the rest of the JIT execution. The cache behavior of the translate portion is illustrated in Figure 3.5. The translation related instruction cache misses contribute to around 30% (except *jack* and *mtrt*) of all the instruction cache misses. Also, it was found that the instruction cache locality is better within the translate routines (miss rate of 1.1% for *db*) than the rest of the JIT (miss rate of 1.5% for *db*). This is due to the high code reuse exhibited by the code generation routines within translate. The same methods are invoked for translating specific bytecodes that occur many times. On the other hand, the data cache misses of the translate routines do not exhibit any general trends, and are dependent on the application. For *mpeg*, *compress* and *db* benchmarks, the data cache exhibits a better locality in the code outside the translate routine. While *compress* benefits from high spatial locality operating on sequential elements of large files, *db* benefits from reuse of a small database to perform repeated data base operations. For *javac*, it was found that the code within and outside translate exhibit a similar cache behavior (miss rates of 5.5 and 5.3% inside and outside translate). This can be ascribed to *javac* being a complier and the executed code performing the same type of operations as the translate routine.

The data cache misses in the translate portion of the code contribute to 40-80% of all data misses for many of the benchmarks. Among these, the data write misses dominate within the translate portion and contribute to 60% of misses during translate (see the third bar for each benchmark in Figure 3.5). Most of these write misses were observed to occur during the generation and installation of the code. Since, the generated code for the method is written to memory for the first time, it results in compulsory misses in the D-Cache. One may expect similar compulsory misses when the bytecodes are read during translation. However, they are relatively less frequent than the write misses since 25 native (SPARC) instructions are generated per bytecode on the average [62]. An optimization to lower the penalty of write misses during code generation and installation is discussed in Section 3.3.

We also studied the variation in the cache locality behavior during the course of execution for different benchmarks in the interpreter and JIT compiler modes. The results for *db* can be observed in Figures 3.6 and 3.7. The

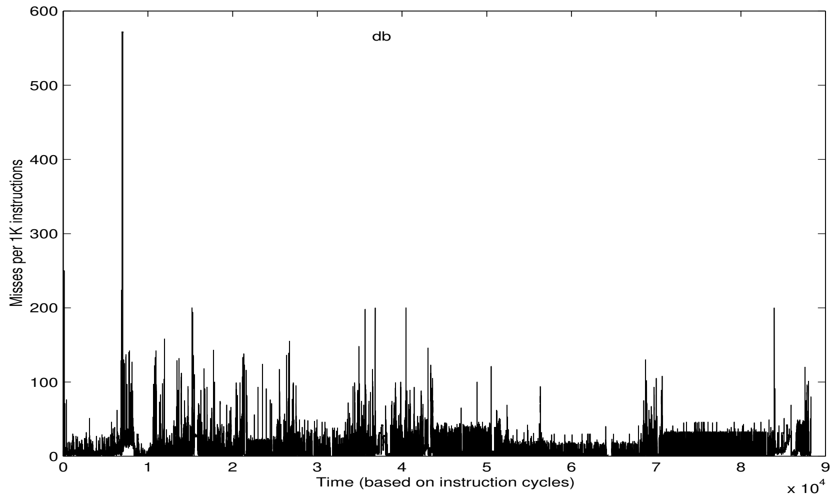

Figure 3.6: Miss rate variation in D-Cache for *db* during code execution in Interpreter Mode. Cache configuration used : 4-way set associative, 64K Cache with a line size of 32 bytes.

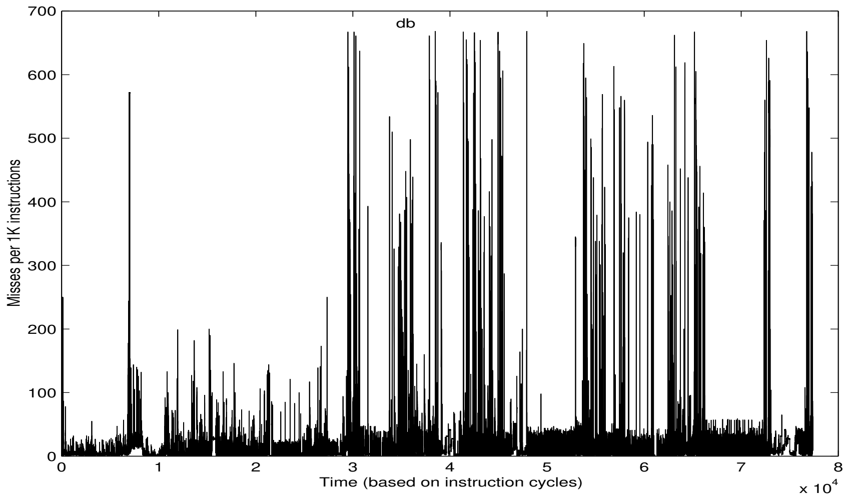

Figure 3.7: Miss rate variation in D-Cache for *db* during code execution in JIT Compiler Mode. Cache configuration used : 4-way set associative, 64K Cache with a line size of 32 bytes.

miss rates in the interpreter mode (Figure 3.6) show initial spikes due to the class loading at the start of the actual execution. However, there is a fairly consistent locality for the rest of the code. In contrast, there are a significantly larger number of spikes in the number of misses during execution in the JIT mode (Figure 3.7). This can be attributed to the compilation part of the JIT compiler which results in significant number of write misses. A clustering of these spikes can be observed in the JIT mode in Figure 3.7. This is due to a group of methods that get translated in rapid succession. Also, we observed that for the *mpeg* benchmark the clustered spikes in the JIT mode are restricted to the initial phase of algorithm as there is significant reuse of the same methods.

3.2.4 Other Observations From Cache Studies

The cache performance of SPECjvm98 applications is studied over a wide range of cache sizes, block sizes and associativity. Figures 3.8 and 3.9 illustrate that increasing associativity produces the expected effect of reducing misses, and the most pronounced reduction is when associativity is increased from 1 to 2. Increasing the line size also produces the usual effect of reducing cache misses in instruction caches, however, data caches display a different behavior (illustrated in Figures 3.10 and 3.11).

For interpreted code, in 6 out of the 7 benchmarks, a small data cache block size of 16 bytes is observed to have the least miss rate for the data cache. On the other hand, for execution with the JIT compiler, a block size of 32 or

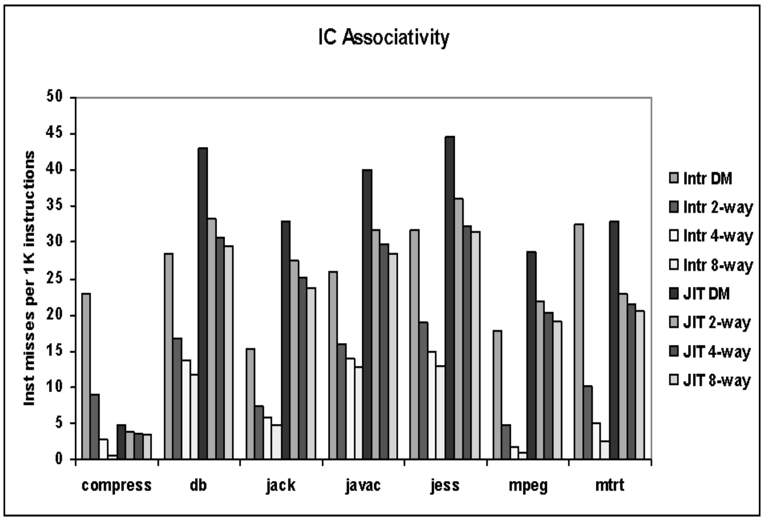

Figure 3.8: Effect of increasing the associativity of the instruction cache. Cache configuration used: 8K Cache with 32 byte line size and associativity of 1, 2, 4 and 8.

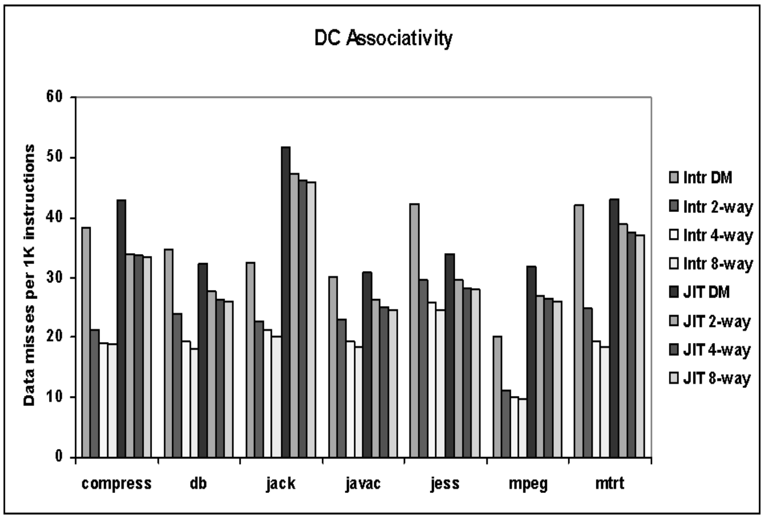

Figure 3.9: Effect of increasing the associativity of the data cache. Cache configuration used: 8K Cache with 32 byte line size and associativity of 1, 2, 4 and 8.

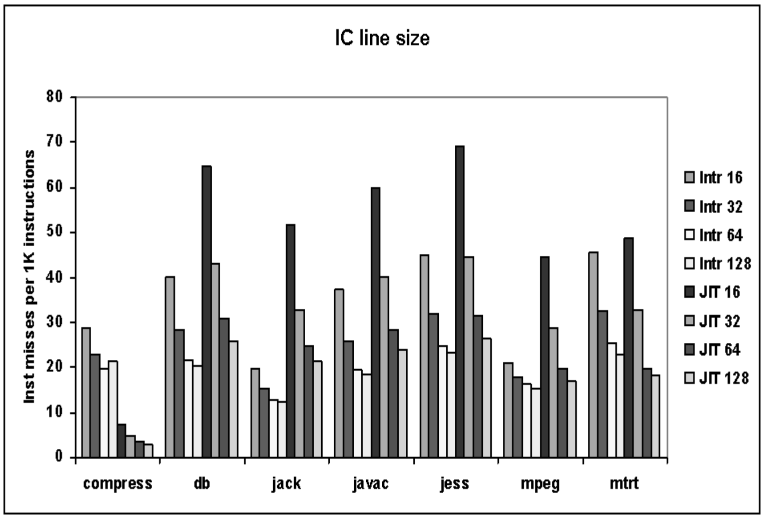

Figure 3.10: Effect of changing line size of the instruction cache. Cache configuration used: 8K direct mapped Cache with line sizes of 16, 32, 64 and 128 bytes.

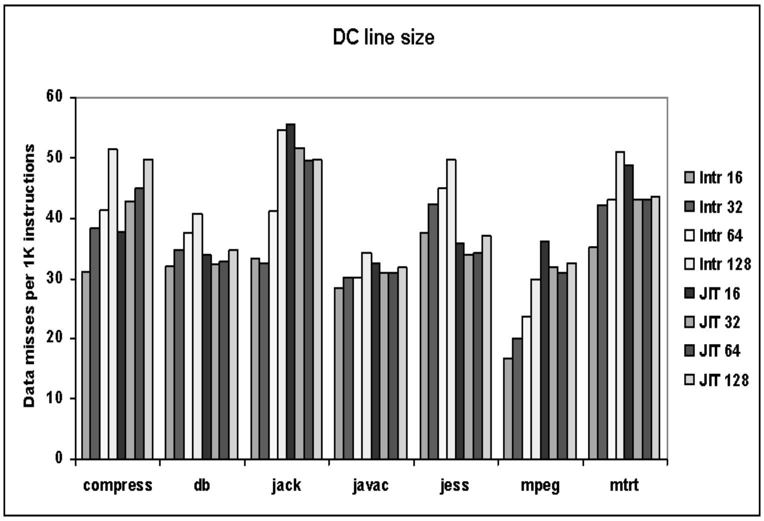

Figure 3.11: Effect of changing line size of the data cache. Cache configuration used: 8K direct mapped Cache with line sizes of 16, 32, 64 and 128 bytes.

64 bytes is better than 16 bytes in a majority of the cases. The increase in data cache miss rates when the line size is increased can be explained using method locality and bytecode size information. Prior research on method locality and size distribution [62] showed that 45% of all dynamically invoked methods were either 1 or 9 bytecodes long. Since average bytecode size has been shown to be 1.8 bytes [78], 45% of all methods can be expected to be less than 16 bytes long. Therefore, unless methods invoked in succession are located contiguously, increasing line sizes beyond 16 bytes (or 32 at the most) cannot capture further useful future references, explaining the data cache behavior of the interpreted code. The data cache in the JIT compiler mode is affected by the size of the objects accessed by the applications. While mean object sizes of individual objects range from 16 to 23 bytes for the SPECjvm98 benchmarks, the commonly used character arrays range between 26 and 42 bytes [23]. Thus, line sizes of either 32 or 64 bytes provide the best locality for most of the benchmarks.

Layout of translated code installed by the JIT compiler can have a large impact on miss behavior. We are not aware of the details on the techniques used by Kaffe or JDK to optimize code layout. Dynamically generated code layout can thus be an interesting area for further research.

3.2.5 Instruction Level Parallelism

Instruction level parallelism (ILP) support is becoming increasingly predominant in current microprocessors. An investigation of ILP issues for Java runtime systems can not only enlighten us on the suitability of this support on general-purpose microprocessors for Java programs, but can also help us incorporate such support in specialized Java processors. It should be remembered that these general purpose processors have not really been tuned for the object-oriented nature of Java or the stack architecture of the JVM. In order to draw some insight into ILP related issues, we have simulated the execution of SPECjvm98 programs on a superscalar processor simulator configured to simulate a 4-way, 8-way and 16-way machine.

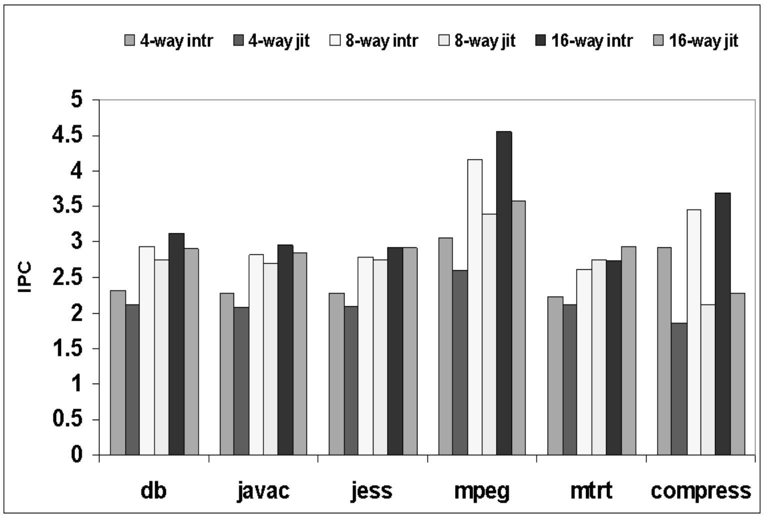

Figure 3.12: Instruction execution rate of the SPECjvm98 benchmarks

Figure 3.12 illustrates the rate of execution of the benchmarks in interpreted and JIT compiler modes and points to the available parallelism in the programs. A noticeable observation is that the instructions executed per cycle (IPC) is higher in the interpreter mode, when compared to the JIT compiler mode. We attribute this to two reasons: (i) the superior cache performance of interpreter mode compared to JIT compiler mode (as observed in Section 3.2.3) and (ii) the inherent unoptimized nature of interpreted code which leads to significant overlap and concurrency while using dynamic scheduling and optimization techniques in modern microprocessors. As observed in Section 3.2.2, the JIT compiler has a superior branch predictor performance compared to the interpreter. This enables the JIT to overcome some of the deterioration in IPC performance caused by poor cache performance. Hence, the JIT compiler mode achieves an IPC not significantly worse than that in interpreter mode.

While the absolute IPC of the interpreter mode is better, the progressive improvement with larger issue processors becomes smaller for the interpreter as can be observed for *jess* and *mtrt* in Figure 3.12. These results can be attributed to the poor target prediction for the switch construct used in the interpreters. This construct is used to jump to the corresponding interpretation code for the bytecode to be executed. The IPC is initially higher for the interpreter as the instructions for interpreting a single bytecode could be optimized by removing false dependencies imposed by the stack based processing. However as issue width increases some of the simple bytecodes such as *iadd*, *isub* [45] would be completely executed in a cycle and fetching the next bytecode is a bottleneck. This bottleneck is caused by the inaccuracy in predicting the target of the switch construct. This observation indicates an interesting possibility for improving the structure of the interpreters. The picoJava processor [53] employs a folding optimization in which commonly occurring sequences of bytecodes of length 2, 3, and 4 are combined in a single execution cycle. An interpreter code that identifies these sequences of bytecodes can mitigate the effect of inaccurate target prediction and scale better. We also expect the scaling of interpreters to improve with architectural support features such as trace caches to exploit cache locality of the instructions and indirect branch predictors to improve the target prediction of switch constructs. This raises an interesting question: Can we actually do more efficient software interpretation with these enhancements as compared to hardware interpretation performed by stack-based Java Processors?

3.2.6 Limits of Available Parallelism

In order to understand the instruction level parallelism issues involving the stack-oriented Java code, we investigated the limits of available parallelism in Java workloads. We also compared the ILP of the Java benchmarks to SPEC95 applications, and several C++ programs. This work is focused on logical limits rather than implementation issues and towards this end, we include an ideal machine with infinite machine level parallelism (MLP).

We use dynamic dependence analysis in order to compute the limits of ILP as in previous parallelism investigations [14, 66]. First, we construct a Dynamic Dependency Graph (DDG) which is a partially ordered, directed, and acyclic graph, representing the execution of a program for a particular input. The executed operations comprise the nodes of the graph and the dependencies realized during the execution form the edges of the graph. The edges in the DDG force a specific order on the execution of dependent operations - forming the complete DDG into a weak ordering of the programs required operations.

A DDG, which contains only data dependencies, and thus is not constrained by any resource or control limitations, is called a dynamic data flow graph. It lacks the total order of execution found in the serial stream; all that remains is the weakest partial order that will successfully perform the computations required by the algorithms used. If a machine were constructed to optimally execute the DDG, its performance would represent an upper bound on the performance attainable for the program. In our study, first, the critical path length defined as the height of the scheduled DDG (the absolute minimum number of steps required to evaluate the operations in the scheduled DDG) is determined. The available parallelism is computed by dividing the total number of nodes by the critical path length.

To give an upper bound on the available parallelism an infinite MLP was considered, however MLPs of 8, 16, 32, 64 and 128 were also studied for comparative purposes. We consider only true dependencies (or RAW dependencies) while scheduling instructions, this is the absolute limit of parallelism that can potentially be exploited from that program, with the best of renaming, etc. The latency of all operations is set to be 1 cycle. Perfect memory disambiguation and perfect branch prediction are assumed. More details on these experiments can be found in [65]. Table 3.8 shows that Java code exhibits less ILP in comparison to all other workloads analyzed.

With infinite MLP, the mean ILP is 125 for the SPECInt benchmarks and 180 for the C++ programs. The mean ILP is 20 for the Java programs when interpreted and 75 when invoked through the JIT compiler. The ex-

Benchmarks	Available ILP					
SPEC Int95	8	16	32	64	128	Infinity
compress	6.94	13.68	21.9	37.65	65.89	265.01
gcc	7.26	13.94	25.56	40.79	58.15	92.14
go	7.15	14.11	27.26	48.79	70.23	83.81
li	7.47	14.25	25.35	47.17	70.15	72.48
m88ksim	6.45	11.09	21.50	45.59	104.43	133.36
ijpeg	7.44	14.67	28.46	48.01	71.86	8465.51
C++	8	16	32	64	128	Infinity
deltablue	7.28	15.15	30.79	60.48	111.28	270.97
eqn	7.42	14.18	26.54	37.96	43.51	272.09
idl	7.57	14.59	29.05	48.31	63.49	277.01
ixx	7.2	13.83	26.92	51.32	76.05	111.71
richards	7.56	13.78	24.83	35.84	55.99	125.57
SPECjvm98 (Intr)	8	16	32	64	128	Infinity
db	6.75	11.59	19.54	28.55	33.7	41.06
jack	7.02	10.85	19.15	28.87	31.69	34.24
javac	6.66	10.67	15.55	19.82	21.89	25.13
jess	6.69	11.02	16.17	19.54	20.78	22.83
mpeg	7.27	10.11	12.86	14.63	15.19	15.39
mtrt	6.76	10.26	12.92	14.06	14.38	15.82
SPECjvm98 (JIT)	8	16	32	64	128	Infinity
db	6.67	12.31	23.30	43.76	70.46	96.90
jack	7.00	12.99	21.22	40.44	47.88	50.47
javac	6.67	12.22	22.97	42.05	65.33	86.23
jess	6.92	12.78	23.57	39.59	47.72	49.33
mpeg	7.30	13.71	25.12	45.73	76.49	174.10
mtrt	7.04	13.05	23.34	34.24	40.04	86.51

Table 3.8: Available Instruction Level Parallelism of Different Benchmarks (SPECjvm98 Benchmarks are Invoked with an Interpreter and a JIT compiler.)

tremely low ILP of the interpreted Java programs, even with no other control or machine constraints, can be attributed to the stack-based implementation of the Java Virtual Machine (JVM). The stack nature of the JVM imposes a strict ordering on the execution of the bytecodes. The JIT compiler optimizes away most of these stack accesses and convert them to register operations. Hence we see a higher ILP for the same benchmarks when executed using a JIT compiler.

3.3 Architectural Enhancements

We have looked at a spectrum of architectural issues that impact the performance of a JVM implementation, whether it be an interpreter or a JIT compiler. We found that one cannot discount interpretation in an ad hoc manner, since it may be more viable in a resource-constrained (memory in particular) environment. In the following discussion, we briefly summarize our observations, review what we have learned from these examinations, and comment on enhancements for the different runtime systems. More importantly, we try to come up with a set of interesting issues, that are, perhaps, worth a closer look for future research.

An examination of the architectural interactions of the two runtime alternatives, has given us useful insights. It has been perceived that Java (object-oriented programs in general) executions are likely to have substantial indirect branches, which are rather difficult to optimize. While we find this to be the case for the interpreter, the JIT compilers seem sufficiently capable of performing optimizations to reduce the frequency of such instructions. As a result, conventional two-level branch predictors would suffice for JIT mode of execution, while a predictor optimized for indirect branches (such as [27]) would be needed for the interpreted mode.

The instruction level parallelism available in interpreted mode is observed to be higher than while using a JIT compiler. However, lack of predictability of indirect branches resulting from interpreter switch construct affects the performance as one moves to wide superscalar machines. We find that the interpreter exhibits better locality for both instructions and data, with substantial reuse of a few bytecodes. The I-cache locality benefits from the interpreter repeatedly executing the native instructions corresponding to these bytecodes, and D-cache locality is also good since these bytecodes are treated as data.

In general, the architectural implications of a Java runtime system are more dependent on the mode of execution (interpreter or JIT) rather than the object-oriented nature of Java programs. Figure 1.3 shows that a significant component of the execution time is spent in the translation to native code, specifically for applications like *db*, *javac* and *jess*. A closer look at the miss behavior of the memory references of this component in Section 3.2 shows that this is mainly due to write misses, particularly those that occur in code generation/installation. Installing the code will require writing to the data cache, and these are counted as misses since those locations have not been accessed earlier (compulsory misses). These misses introduce two kinds of overheads.

First, the data has to be fetched from memory into the D-cache before they are written into (on a write-allocate cache, which is more predominant). This is a redundant operation since the memory is initialized for the first time. Second, the newly written instructions will then be moved (automatically on instruction fetch operations) from the D-cache to the I-cache (not just causing an extra data transfer, but also potentially double-caching). To avoid some of these overheads, it would be useful to have a mechanism wherein the code can be generated directly into the I-cache. This would require support from the I-cache to accommodate a write operation (if it does not already support it), and preferably a write-back I-cache. It should also be noted that for good performance, one should be careful to locate the code for translation itself such that it does not interfere/thrash with the generated code in the I-cache. A possibility is the reuse of the recently translated code in subsequent translations (so that translation can be speeded up). It was also suggested earlier in Section 3.2 that it may be a worthwhile effort to look into issues of translated code location (perhaps using associations), to improve locality during subsequent executions.

Figure 1.3 shows that there are applications, like *compress*, and *jack*, in which a significant portion of the time is spent in executing the translated code. One possible way of improving these applications, is to generate highly optimized code (spending a little more time to optimize code will not degrade the performance). Another approach is to speed up the execution of the generated code. This could involve hardware and systems software support for memory management, synchronization and class resolution/loading.

There is a common (and interesting) trait in the *compress*, *jack* and *mpeg* applications, where the execution time dominates and a significant portion of this time is spent in certain specific functions. For instance, *compress* and *mpeg* employ a standard set of functions to encode all the data. The benchmark *jack* scans the data, looking for matching patterns. If we are to optimize the execution of such functions, then we can hope for much better performance. A promising research direction is to identify such commonly employed functions (for at least certain application domains), so that one can configure hardware cores using reconfigurable hardware (such as Field Programmable Gate Arrays) on-the-fly (similar to how JIT dynamically opts to compile-execute rather than interpret).

3.4 Summary

This chapter has focused on understanding the influence of hardware-software interactions of the two most common JVM implementations (interpreter and JIT-compiler), towards designing architectural support for efficient execution of Java programs. The major contributions of our research are the following:

1. *Method reuse* is a good heuristic to choose between the interpreter and JIT mode of execution. Applications with low method reuse are executed more efficiently using interpretation, whereas JIT compilers show better performance for applications with high method reuse.

2. Execution of Java applications using software emulation is expensive, requiring the execution of approximately 35 SPARC instructions per bytecode for interpretation, and 20 instructions per bytecode when executed using a JIT compiler.

3. The instruction and data cache performance of Java applications are better compared to that of C/C++ applications, except in the case of data cache performance in the JIT mode.

4. Except using smaller block sizes for data caches or using branch predictors specially tailored for indirect branches, we feel that optimizing caches and branch predictors will not have a major impact on performance of Java execution.

5. Write misses resulting from installation of JIT compiler output has a significant effect on the data cache performance in JIT mode. Certain enhancements, such as being able to write to the instruction cache, could be useful during dynamic code generation.

6. The instruction level parallelism available in interpreted mode is observed to be higher than a JIT compiler, when executed on current superscalar processors. However, lack of predictability of indirect branches resulting from interpreter switch construct affects the performance as one moves to wider superscalar machines.

7. We investigated the limits of available parallelism in Java applications when executed using an interpreter and a JIT compiler, and compared it

to C++ and C applications. The extremely low ILP of Java applications, even with no other control or machine constraints is attributed to the stack based nature of the JVM which imposes a strict ordering on the execution of bytecodes.

The topics that seem to hold the most promise for further investigation are new architectural mechanisms for hiding the cost of translation during JIT compilation. Techniques for achieving this may also be used in conjunction with dynamic hardware compilation (one could visualize this as hardware translation instead of compilation that is done by a traditional JIT compiler) of Java bytecodes using reconfigurable hardware. Another topic that is worth investigating is techniques to extract parallelism in the bytecode stream by alleviating dependencies to the stack, to enable efficient execution of Java applications.

Chapter 4

Translation Support in General-Purpose Processors

Java has an instruction set (bytecodes) that is different from the instruction set of any general purpose processor. Java instructions cannot be executed in their original bytecode form and a translation from the Java instruction set to the instruction set of the processor on the system being executed is required.

Software translation in the form of interpretation or JIT compilation is commonly used to execute Java on general purpose processors. Translation of bytecodes was shown to be the most critical bottleneck when executing Java applications using software emulation in chapter 3. In this chapter, we evaluate the feasibility of using a hardware translator to convert Java bytecodes to the native instructions of the general purpose processor.

4.1 Introduction

Bytecodes have semantics that are different from the semantics of the machine instructions of any general purpose processor. The execution of a bytecode therefore requires the execution of a short sequence of native machine instructions. Software interpretation uses an execution mechanism which consists of repeatedly executing the following steps:

- **step 1.** Locating the next bytecode to be executed, fetching the bytecode and decoding it. This involves:
  - reading and updating a program counter for the bytecodes
  - reading the contents of memory addressed by the bytecode PC
  - decoding the bytecode to get opcode, operands etc
  - transferring control to a routine which corresponds to the opcode just decoded
- **step 2.** Executing the semantics that correspond to the decoded bytecode. This involves executing a short sequence of machine instructions.
- **step 3.** Transferring control back to step 1.

The JIT compiler also uses a similar mechanism, but when it comes across a method call it transfers control to the compiler. The compiler checks if a translated routine exists for the method and if it does not, proceeds to compile the method. In a sense the JIT compiler creates a semantic routine for each method invoked in the Java program at runtime, whereas the interpreter uses a semantic routine for individual bytecodes.

Step 2 actions are called *executive actions* and form the *execution phase* of the software translation. Steps 1 and 3 are responsible for fetching and decode of bytecodes and perform the *mapping actions*. In native compiled code the mapping actions are performed in hardware which is significantly faster. A significant slowdown is observed for software translated code, since the mapping actions are performed in software. We show in chapter 3 that interpretation requires the execution of 35 SPARC instructions on the average to execute each bytecode. For JIT execution 20 SPARC instructions were required on the average to execute each bytecode.

4.2 The Hard-Int Architecture

Our proposal is to use additional hardware in a general purpose processor to perform the mapping actions involved in execution of Java bytecodes. The proposed architecture requires no changes to the core of the general purpose processor. Additional hardware is added at the front end of the processor, thus the native instruction set and pipeline architecture of the processor are unchanged. The additional hardware will have functionality similar to a software interpreter. We use the term Hard-Int (Hardware Interpreter) to denote a general purpose architecture that uses the proposed hardware.

The Hard-Int architecture performs the mapping from bytecodes to native code using hardware, instead of software. The translated instructions are fed to the processor core which executes them in the normal fashion. A high-level block diagram of the Hard-Int system is shown in Figure 4.1. In addition to using a hardware translator, a *translated code buffer* is added to facilitate reuse of the translated bytecodes.

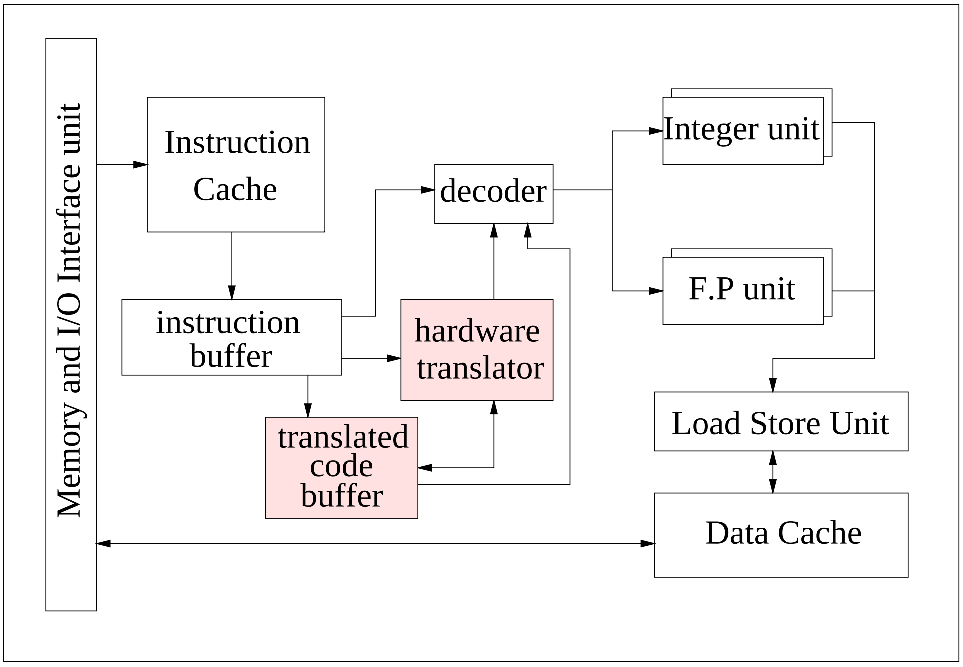

Figure 4.1: The Hardware Interpreter (Hard-Int) architecture
Proposed additions are shaded in grey

As a group of translated bytecodes are being executed by the processor core, a different group of bytecodes bytecodes can be simultaneously fetched and translated by the hardware translator. Thus, system efficiency is greatly improved by overlapping of translation and execution which reduces idle waiting time in the system.

4.2.1 Implementation Details

Figure 4.2 illustrates the proposed Hard-Int units. The hardware translation unit fetches the bytecodes from the instruction cache and predecodes them before performing the translation using the microcoded ROM, which contains the translated code for that bytecode. Complex bytecodes, which form less than 5% of the dynamic instructions executed do not contain entries in the ROM. A dispatch table contains the starting address of corresponding interpreter routine for these complex bytecodes. Software translation is used to execute these less frequently used bytecodes.

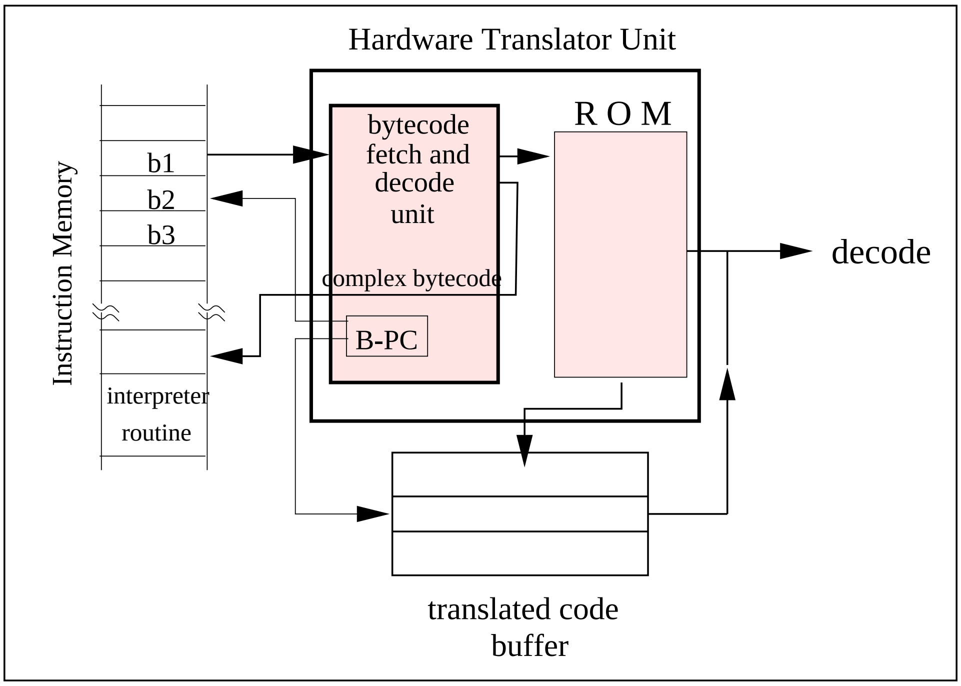

Figure 4.2: Translating bytecodes in the Hard-Int architecture

The bytecode PC is stored in a register (B-PC) and is read when fetching the next bytecode to be executed. After a bytecode is decoded the corresponding native instructions are sent to the decoder from the ROM and are also stored in the translated code buffer. During a fetch, the translated code buffer is checked in parallel with the instruction cache. In the event of a translated code buffer hit, the native instructions are fed to the decoder from the translated code buffer. By providing the instructions from the translated code buffer, an instruction cache access and the hardware mapping (using the microcoded ROM) is bypassed. A translated code buffer allows for further optimization of bytecode sequences, similar to those performed by the JIT compiler.

The proposed architecture is very similar to an instruction path coprocessor [29]. Typically coprocessors have been used in general purpose processors to speedup specific tasks. Coprocessors have been used for performing floating point calculations, graphics and memory management. These coprocessors exist in the data path of the processor and do not help in fetching or decoding instructions. Coprocessors which exist on the instruction path of the processor, obtain instructions from memory, and execute them are called *instruction path coprocessors* [29]. The *instruction path coprocessor* was introduced as a way to provide hardware support to speed up interpretive applications by executing the interpreter mapping concurrently with the CPU using a coprocessor. A programmable instruction path coprocessor is used to perform dynamic transformations to the object code in [18].

The proposed architecture may alternatively be viewed as a translate coprocessor. Along the style of decoupled access-execute architectures [69], it can constitute a decoupled translate-execute architecture [61], with the translate processor performing the conversion from bytecode to native sequence. The execute processor is responsible for only the executive actions pertaining to the program, whereas the translative or mapping actions corresponding to the program are handled by the translate processor.

Examples of bytecode translation

We provide a few simple examples of the translation of bytecodes to native code (with SPARC as the native ISA.) We will translate the following bytecodes: (i) *swap* which swaps the top two words on top of the operand stack, (ii) *dup* which duplicates the top word on the operand stack and pushes it into the operand stack and (iii) *lstore* which stores the word on the top of the operand stack, of type *long* into the local variable area. The offset into the local variable is obtained from an unsigned byte following the *lstore* opcode.

The mapped instructions that are stored in the microcoded ROM are shown in SPARC assembly, and variable names are used in place of explicit register numbers. Variables used here are: (i) SP - pointer to the topmost entry of the operand stack (ii) TOSA and TOSB - registers to hold temporary computation values, (iii) NextByte - parameter for current bytecode (obtained by doing a fetch ahead of the instruction stream) and (iv) Varbase - pointer to array of local variable. Table 4.1 shows the mapping scheme used to translate *swap*, *dup* and *lstore*. A similar mapping scheme is used for most of the bytecodes and the SPARC instructions are stored in the microcoded ROM.

Examples of translation routines used for *swap*, *dup* and *lstore*

Bytecode	SPARC ops	comments
swap	ld [SP], TOSB	;load TOS entry to register
	ld [SP-4], TOSA	;load TOS-1 entry to register
	st TOSB, [SP-4]	;swap entries in the stack
	st TOSA, [SP]	
dup	ld [SP], TOSA	;load TOS into register
	st TOSA, [SP+4]	;store register value to TOS+1
	inc 4, SP	;increment stack top pointer
lstore	ld [SP-4], TOSA	;load TOS-1 entry to register
	sll NextByte, 2, T2	;get offset value from NextByte, store in register
	dec 4, SP	; decrement stack pointer
	ld [SP+4], TOSB	;load TOS into register
	st TOSA, [Varbase+T2]	;store 1st register value to local variable area
	add Varbase, 4, T1	;point to next location
	st TOSB, [T1+T2]	; store 2nd register value to local variable area

Table 4.1: Examples of bytecode to SPARC assembly code translations

4.3 Experimental Methodology

The performance of the Hard-Int architecture is evaluated by comparing it to different JVMs running on a 4-way superscalar processor machine as well as an aggressive 16-way superscalar processor machine. A detailed, cycle-accurate microprocessor simulator using Shade [19] as the functional execution engine is used to compare the performance of the Hard-Int architecture against the Sun JDK 1.1.6 and Sun JDK 1.2 JVMs (which are run using an interpreter and JIT compiler). The runtime environment for JDK 1.2 is more optimized compared to JDK 1.1.6 and uses more efficient synchronization and garbage collection. The JIT compiler used in JDK 1.2 is more mature compared to the JDK

1.1.6 JIT, performing more optimizations and selective compilation. Table 4.3 shows the configuration of the 4-way superscalar and 16-way superscalar machines used in this study.

To model Hard-Int architecture, the simulator is modified to incorporate the hardware translator and a table lookup scheme to decode the bytecodes. A mapping from bytecode to SPARC native instructions as described in the previous section is used to emulate the bytecodes. The mappings are stored in a microcoded ROM and a latency of 3 cycles is attributed to decode the bytecode and fetch the corresponding mapping from the ROM. A 64-entry, 4-way translated code buffer with an access time of two cycles is used to cache the SPARC instructions associated with recently decoded bytecodes. Each line of the translated code buffer can hold up to 16 SPARC instructions associated with one or more bytecodes. If a bytecode maps to more than 16 SPARC instructions, the translated code buffer is accessed in the next cycle to retrieve the remaining instructions. In the event of a translated code buffer miss, the bytecodes are fetched from the instruction cache and decoded before obtaining the corresponding entries from microcoded ROM. This incurs the latency of an instruction cache access in addition to three cycles to access the microcoded ROM. The rest of the configuration of the Hard-Int architecture is kept the same as in Table 4.3.

DATA MEMORY

	Configuration-1	Configuration-2
- L1 Data Cache:	4-way, 64KB, 1-cycle access	4-way, 64KB, 1-cycle access
- L2 Unified cache:	4-way, 1 MB, +7 cycles	4-way, 1 MB, +7 cycles
- Non-blocking	2 MSHRs and 1 port	8 MSHRs and 4 port
- D-TLB	128-entry, 1-cycle hit, 30-cycle miss	128-entry, 1-cycle hit, 30-cycle miss
- Store buffer:	32-entry w/load forwarding	128-entry w/load forwarding
- Main memory:	Infinite, +22 cycles	Infinite, +22 cycles

FETCH ENGINE

	Configuration-1	Configuration-2
- L1 Instruction Cache:	4-way, 64KB, 1-cycle hit	4-way, 64KB, 1-cycle hit
- Branch Predictor:	16k gshare predictor 8-cycle misprediction penalty	16k gshare predictor 8-cycle misprediction penalty
- Indirect Branch target buffer	512 entries	512 entries

EXECUTION CORE

	Configuration-1			Configuration-2		
Int/FP ROB entries	64 each			144 each		
Decode Width	4 instructions			16 instructions		
Issue Width	4 instructions			16 instructions		
Execute Width	4 instructions			16 instructions		
Retire Width	4 instructions			16 instructions		
- Functional unit	#	exec. lat.	issue lat.	#	exec. lat.	issue lat.
Load/store	2	1 cycle	1 cycle	6	1 cycle	1 cycle
Simple Integer	3	1	1	9	1	1
Int. Mul/Div	2	3/20	1/19	6	3/20	1/19
Simple FP	2	3	1	4	3	1
FP Mul/Div/Sqrt	2	3/12/24	1/12/24	4	3/12/24	1/12/24

Table 4.2: Configurations of simulated processor

4.4 Performance Evaluation

The performance of the proposed model is compared to both the interpreter and JIT execution modes using JDK 1.1.6 and JDK 1.2 JVM runtime environments. However, we do not include static native compilers in the comparison study. Static compilers, which compile Java to native code are used by a very small segment of Java users. This is because the current generation of static bytecode compilers does not support the entire JVM specification. Also, it has been our experience that using a static compiler is cumbersome, and does not provide significant performance gains, if any, over sophisticated JIT compilers. TowerJ [8] is the most stable and commonly used commercial static compiler available today, but it does not provide runtime support for the execution of the SPECjvm98 benchmarks which is used in this dissertation. Due to the aforementioned reasons we preclude static compilers in the performance evaluation study conducted in this chapter.

4.4.1 Execution speedup

Figure 4.3 shows the number of execution cycles needed to run the SPECjvm98 benchmarks on a 4-way processor machine for the two interpreters, the two JIT compilers and the Hard-Int architecture. The Hard-Int architecture performs better than the interpreter for all the benchmarks, resulting in speedups of 2.81 for *db* to 7.7 for *jess*. The translation overhead for a bytecode in the Hard-Int architecture is smaller than the overhead incurred in interpreters. The interpreter must fetch each bytecode from the data cache, and perform certain bookkeeping tasks (like incrementing the PC and stack pointers) before executing each bytecode. By lowering the translation overhead for executing each bytecode, the Hard-Int architecture performs better than the interpreter for all the benchmarks.

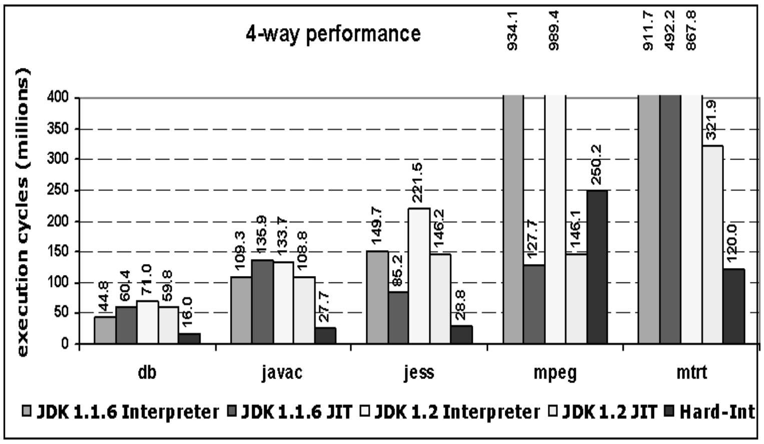

Figure 4.3: Execution cycles for different execution modes on a 4-way machine

On comparing the performance of the Hard-Int architecture to that of the JIT compiler, Hard-Int provides speedup ranging from 2.68 for *mtrt* to 5.1 for *jess*, except in the case of *mpeg*. For the *mpeg* benchmark, the JIT compiler has a speedup of 1.72 over the Hard-Int architecture. The JIT compiler performs better for this benchmark since most of the execution time is spent in a few methods which were compiled by the JIT compiler during the initial phases. The JIT compiler obtains a speedup of 7.3 over the interpreter for *mpeg* illustrating the exploitation of method reuse.

The high speedup obtained by Hard-Int for *db*, *jess* and *javac* shows that the Hard-Int model performs consistently better for client workloads, where the JIT results in little or no improvement over an interpreter. For certain long-running workloads (like *mpeg*) the Hard-Int model performs better than the interpreter. However, there is performance degradation when compared to a JIT compiler. The performance of the Hard-Int architecture can be improved further by lowering the overhead associated with the fetch and decode of bytecodes. A hardwired decoder for the simpler and more frequently executed bytecodes will reduce the decode overhead to one cycle. Using a bigger translated code buffer and allowing for optimizations on the stored bytecodes will result in generation of more optimized sequences of bytecodes, similar to the code generated by JIT compilers.

Figure 4.4 shows the number of cycles executed for the benchmarks when run on a 16-way machine. The Hard-Int architecture outperforms both interpreters in all cases, with speedups in the 2.86 to 7.57 range. There is improvement over JITs in all benchmarks except *mpeg*. The average speedup (geometric mean) over the five benchmarks is 2.26 (over JDK 1.2 JIT), 2.07 (over JDK 1.1.6 JIT), 5.07 (over JDK 1.2 interpreter) and 3.95 (over JDK 1.1.6 interpreter). The current implementation of the hardware translator converts one or more bytecodes at a time, and stores the translated instructions in the translated code buffer. If we store traces of bytecodes, similar to a trace cache, we can improve the ILP and increase performance when going to wider-issue machines. Another option to improve performance is to store the translated code for methods together, and fetch them on a method call which will increase the ILP and help the Hard-Int architecture to be effective in wide issue machines.

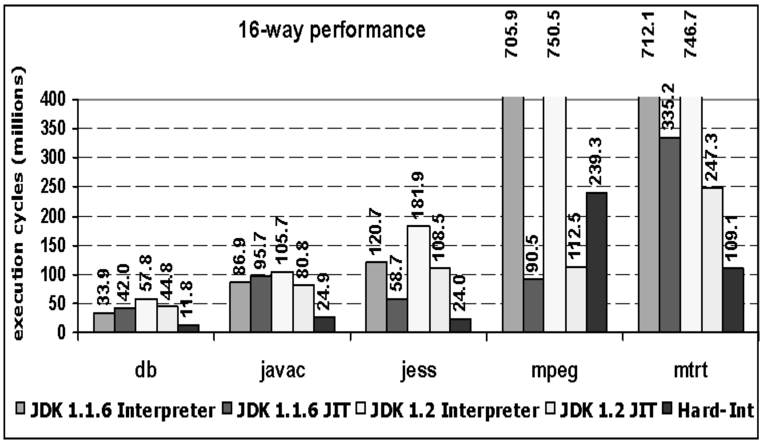

Figure 4.4: Execution cycles for different execution modes on a 16-way machine

Caveats

The Hard-Int architecture is seen to perform better than JIT compilers for certain benchmarks such as *db*, *jess* and *javac*. On the other hand, the JIT shows better performance than the Hard-int for *mpeg*. The Hard-Int architecture presented in this dissertation performs a simple table lookup and we have not implemented optimizations on the translated code. A JIT that can perform optimizations over multiple bytecodes can generate high quality code and can excel Hard-Int once the translation overhead is amortized. A detailed study using different benchmarks and varying run lengths is required to better understand the JIT overhead amortization point and its correlation to performance of hardware translation.

4.4.2 Cache performance

The cache performance of the SPECjvm98 benchmarks for the various execution modes is shown in Table 4.3. When using a hardware translator the bytecodes are stored and fetched from the instruction cache and the translated code is written to the translated code buffer. We see better instruction and data cache performance using this approach, since we do not store the bytecodes nor the translated code to the data cache as in the JVM. Fewer instruction and data cache references are generated when using the Hard-Int architecture.

There are fewer cache misses in a majority of cases, except a few benchmarks where the JDK 1.1.6 interpreter shows impressive instruction cache performance. The interpreter executes a basic loop that fetches and decodes a bytecode fetched from the data cache. The instruction footprint of the interpreter is small and we see very few misses in the instruction cache. The JIT compiler on the other hand has a larger instruction footprint than the interpreter and also generates references to the compiled native code. The working set switches between the compiled code and the code for the JIT compiler often, resulting in more instruction cache misses.

The difference in number of misses is more prominent in the case of the data cache, where the JIT compiler has higher misses that the interpreter for all the benchmarks. This is the case even when the interpreter has more data references in certain benchmarks such as *mtrt* and *javac*. Loads to fetch bytecodes are the main source of data cache references for the interpreter. In

Benchmark	JVM	Execution mode	I-Cache Refs	I-Cache misses	D-Cache Refs	D-Cache misses
db	JDK 1.1.6	interpreter	13.52M	4304	21.9M	225425
	JDK 1.1.6	JIT	19.37M	125768	27.67M	454032
	JDK 1.2	interpreter	19.51M	77772	35.03M	440581
	JDK 1.2	JIT	17.50M	189801	28.69M	615089
	Hard-DTE		7.62M	35820	11.91	299405
jess	JDK 1.1.6	interpreter	40.36M	17012	73.54M	791518
	JDK 1.1.6	JIT	29.48M	140802	43.09M	885864
	JDK 1.2	interpreter	55.79M	264831	107.4M	1.53M
	JDK 1.2	JIT	42.43M	634877	67.23M	1.88M
	Hard-DTE		10.10M	36659	11.98M	314735
javac	JDK 1.1.6	interpreter	31.01M	14058	53.71M	596469
	JDK 1.1.6	JIT	41.82M	265825	61.16M	1.08M
	JDK 1.2	interpreter	35.82M	205450	61.31M	979362
	JDK 1.2	JIT	31.81M	456543	48.07M	1.24M
	Hard-DTE		13.2M	54169	17.62M	439674
mpeg	JDK 1.1.6	interpreter	206.73M	12314	451.62M	511866
	JDK 1.1.6	JIT	37.23M	197381	91.45M	765343
	JDK 1.2	interpreter	222.47M	125611	485.7M	917192
	JDK 1.2	JIT	37.7M	302112	110.68M	1.01M
	Hard-DTE		36.03M	77292	54.24M	522781
mtrt	JDK 1.1.6	interpreter	218.35M	17162	487.38M	5.15M
	JDK 1.1.6	JIT	162.69M	172562	229.64M	5.56M
	JDK 1.2	interpreter	189.17M	105299	502.25M	4.06M
	JDK 1.2	JIT	81.9M	266588	164.41M	4.50M
	Hard-DTE		17.82M	76356	58.21M	518483

Table 4.3: Cache Performance for the SPECjvm98

This table shows the number of references and misses for the instruction and data cache. Cache size= 64K bytes, block size= 32 bytes, I-cache is 2-way and D-cache is 4-way set-associative. M - indicates million.

the JIT compiled execution, the compiled code has to be written to memory and cause write misses in the data cache. When that code is executed for the first time it also causes misses in the instruction cache.

Benchmark	References	Hits	Hit Rate
db	13.28M	927392	69.79%
jess	5.40M	4.39M	81.34%
javac	3.86M	2.86M	74.32%
mpeg	72.97M	55.77M	76.42%
mtrt	34.94M	19.11M	54.70%

Table 4.4: Translated code buffer performance for the SPEcjvm98

This table shows the number of references and hits for the translated code buffer. Buffer size is 64 entries and 4-way set-associative. `M indicates million.`

4.4.3 Performance of the translated code buffer

The translated code buffer is a 64-entry, four-way associative buffer which is used to store the corresponding SPARC instructions associated with recently executed bytecodes. It is a means of enabling some reuse at the bytecode level. A JIT compiler stores the translated code for methods, and thereby making it unnecessary to translate that method when it is invoked in the future. Similarly, but at a finer granularity, we store the translated code for each bytecode in a buffer and provide the translation from the buffer if the bytecode is executed again. If the translated code for a bytecode does not exist in the translated code buffer, the mapping is provided by the microcoded ROM after the bytecode is fetched and decoded.

Table 4.4 shows the references and misses encountered by the translated code buffer. Since we use a small buffer of only 64 entries, a hit rate of 54.7% to 81.3% is observed. The translated code buffer used here is a very simple structure. Optimizations can also be done on the stored entries in the translated code buffer which will allow the performance of the hardware translator to be more at par with the JIT performance for server workloads.

4.4.4 Emulation cost

The larger the architectural gap that exists between the JVM and the emulating processor, the higher the number of cycles taken to emulate a bytecode. Figure 4.5 shows the average number of cycles to emulate (translate and execute) a single bytecode on a 4-way processor.

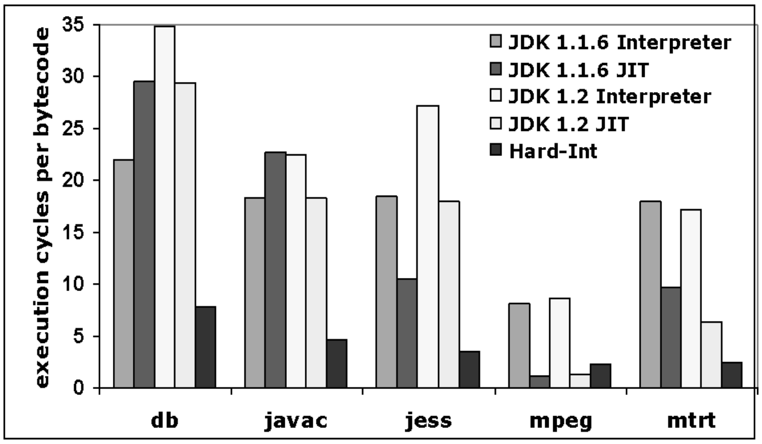

Figure 4.5: Cycles executed per bytecode on a 4-way machine

Using an interpreter, the overhead associated with fetching and decoding each bytecode in software is high resulting in a large translation cost ranging from eight cycles per bytecode executed (for *mpeg*) to 35 cycles per bytecode (for *db*). In the Hard-Int architecture the fetching and decoding of the bytecodes are done in hardware (taking two cycles for each bytecode). The cost seen in Figure 4.5 ranges from 2.2 cycles for *mpeg* to around eight cycles for *db*. The emulation cost for the JIT ranges from 1.1 cycles for *mpeg* to 29.7 cycles for *db*. The cost is low for benchmarks which spend lot of time in a few methods, and higher if the execution time is distributed across a lot of methods. The JIT does a good job of exploiting the architectural features using optimizations, but the compilation time is added to the translation overhead. If the compiled code is not executed frequently and more time is spent in compiling methods than executing them, the cost may be higher than when executed using an interpreter. This is observed in Figure 4.5 for *db* and *jess* when executed using the JDK 1.1.6 interpreter and JIT.

4.4.5 Performance bottlenecks

Figure 4.6 graphically illustrates the retire-time bandwidth for each benchmark. In the 4-way configuration, four instructions can potentially be retired each cycle. This figure breaks down the utilization of these retire slots. The bottom section of the column is the percentage of retire slots that are filled with instructions that successfully retire. The section above that represents the percentage of slots that are empty due to instruction cache misses. The next portion of the bar (and generally the smallest for these benchmarks) is due to store instructions that can not progress due to memory resource contentions. The load stalls segment is above the store stalls. Loads stalls are caused by load instructions that are either currently in the process of loading or can not progress due to memory resource contentions. The slice second from the top is due to pipeline stalls. These instructions are detained because the pipeline is backed up and they have yet to execute. The top section of the column represents retire-time bandwidth lost due to branch mispredictions.

When a branch mispredicts, no other valid instructions can reach the retire slots until the branch has resolved and a new fetch has occurred.

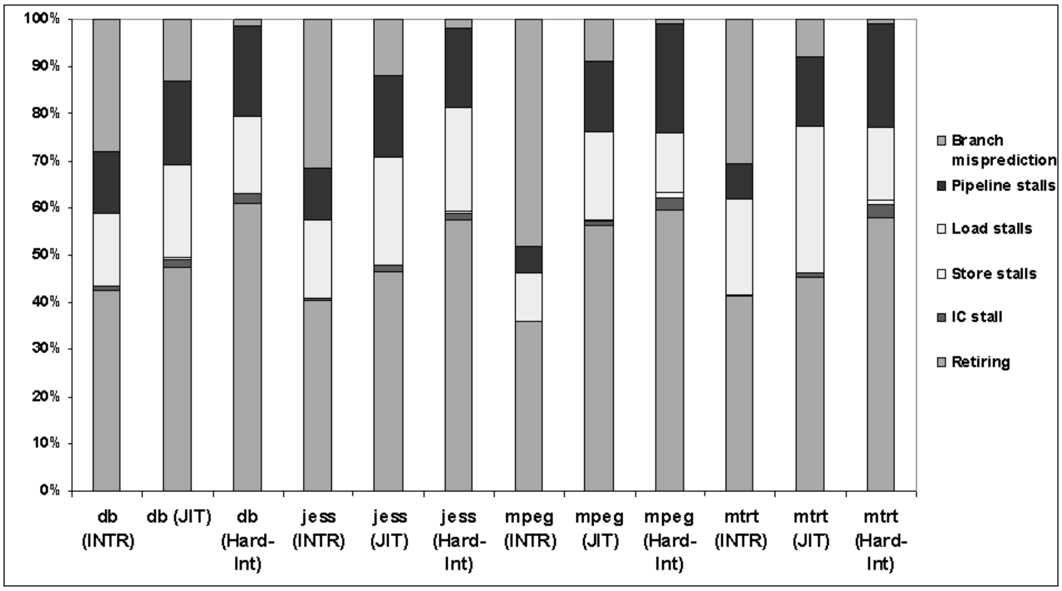

Figure 4.6: Breakdown of IPC for JDK 1.2 and HardInt

The top slice of the breakdown, which shows the stalls due to branch misprediction is the most striking difference between the different execution modes. The performance of the branch predictor is fairly consistent across the different modes, but it is the number of indirect branches and BTB misses which account for the difference. The interpreter executes a large number of indirect branches to implement the decode loop and large number of BTB misses occur when predicting the targets. This effect is mitigated in the JIT but not completely eliminated. The Hard-Int does not execute a branch to decode the bytecode and branch misprediction forms a small percent of the retire bandwidth. Stalls due to store waiting are more prominent for the Hard-Int architecture. Benchmarks *mtrt* and *mpeg* show up to 1.3% stalls due to stores that could not complete. This is due to memory operations becoming a more prominent percentage of the instruction mix in the Hard-Int architecture. The percentage of load stalls are similarly higher in the Hard-Int architecture compared to the interpreter. The load stalls are seen to be fairly large across all three execution modes and is the worst for the JIT, where the stall is as high as 32% in *mtrt*. The pipeline stalls are more of a bottleneck in performance for the Hard-Int architecture since other stalls are not as prominent in the other execution modes. It is evident that eliminating BTB misses mode would result in substantial performance improvement for the interpreter. Load stalls are seen to limit the JIT performance, and in the case of the Hard-Int architecture it is mainly memory performance which is the bottleneck.

4.5 Summary

The primary bottleneck in Java execution is the overhead of translation. In interpretation, it is the overhead of software fetch and decode. In JITs, it is the overhead of (i) performing compilation and compiler optimizations at run time, (ii) installing the compiled code into data caches/memory at run time, and (iii) using extra memory for the JIT itself. The overheads are more prominent for short-running, client-like workloads, where the execution time is not concentrated in a small number of methods. Using additional hardware in general purpose processors to perform the *mapping actions* lowers the overhead and improves performance of client-like workloads. We proposed the Hard-Int architecture which dynamically translates bytecodes to native machine instructions using a hardware translation unit and a translated code buffer. The hardware translation unit fetches the bytecodes from the instruction cache and predecodes them before performing the translation using the microcoded ROM. The proposed architecture does not require changes to the core of the general purpose processors, thus enabling use of existing expertise and legacy software.

We compared the performance of SPECjvm98 benchmarks when executed using software translation (interpreter and JIT) to hardware translation using the Hard-Int architecture.

We summarize the important observations below:

- Hardware translation results in an average[1] speedup of 5.43 (4.39) over the JDK 1.2 (JDK 1.1.6) interpreter on a 4-way processor. The average speedup attained in a 16-way processor is 5.07 (3.95).

- The Hard-Int architecture performs consistently better than a JIT compiler for client workloads, showing an average speedup of 2.59 (2.58) over the JDK 1.2 (JDK 1.1.6) JIT compiler on a 4-way machine. The average is significantly affected by *mpeg*, which is the only program that shows a performance degradation. Excluding *mpeg*, the average speedup is 3.76 (3.87) over the JIT compiler on a 4-way machine. On a 16-way machine, the Hard-Int architecture shows a speedup of 2.26 (2.07) over the JIT compiler. With *mpeg* excluded, the average speedup obtained is 3.35 (3.18) on a 16-way machine.

[1] all averages for speedup use the geometric mean

- The emulation cost per bytecode varies from 2.2 cycles per bytecode to 7.8 cycles per bytecode for the SPECjvm98 benchmarks when executed on a 4-way processor using the Hard-Int architecture. In the interpreter the cost ranged from 8.1 to 34.9 cycles per bytecode, and from 1.1 to 29.7 in a JIT.

- When analyzing performance bottlenecks, we find that the cache performance of the Hard-Int is better than the JIT. This is because the translated code is not written to memory, and therefore write misses in the data cache and read misses in the instruction cache are not incurred as frequently. In addition, Hard-Int results in less speculative execution due to fewer branch direction and target mispredictions.

Current execution modes of Java using software translation suffers from poor performance compared to compiled native code. Providing hardware support in general purpose processors is one way to bridge this performance gap between the execution of Java applications and natively compiled applications. This chapter shows that the existing performance gap can be greatly reduced using the proposed Hard-Int architecture.

Chapter 5

Exploiting Parallelism in Embedded Java Processors

Java processors such as Sun Microsystems picoJava cores [50, 53] and JEM [82] are low-cost hardware engines optimized to directly execute Java bytecodes. The bytecode ISA is stack based and provides the advantage of dense code, which is important for the embedded environment. However, direct execution of bytecodes on stack based embedded processors is invariably constrained by the limitations of the stack architecture for accessing operands. The parallelism study at the end of Chapter 2 gives a major conclusion that rather than providing dedicated hardware to execute stack based code, it is beneficial to perform optimizations that convert stack based code to register based code in order to exploit ILP. In this chapter we investigate techniques that allow to exploit parallelism in the bytecode instruction stream, and thus improve the performance of embedded Java processors.

5.1 Introduction

Folding [44] is an optimization implemented in stack architectures (for example, picoJava-I and picoJava-II) to coalesce multiple stack based instructions to a single RISC-style instruction with optimized data accessing. High end Java platforms such as servers can implement equivalent optimizations in the JIT compiler. Microarchitectural enhancements in modern ILP processors can also contribute to exploiting the parallelism in Java. Space-time computing and multithreaded execution on clustered hardware as in Sun's MAJC [48] architecture will exploit both ILP and thread level parallelism in Java. However, at the low end of the Java horizon, Java chips for embedded and networked arena need to be specially tailored to exploit fine-grain parallelism. Many ILP techniques used at the server end will not be feasible to implement at the embedded end.

The basic goals of this chapter are to investigate cost-effective microarchitectural techniques to exploit the available parallelism in the bytecode stream and enhance the performance of Java processors for the embedded environment. We explore these issues adopting the picoJava-II processor core from Sun Microsystems as the reference processor. This choice is well justified as the picoJava-II core implements several optimizations including folding and is the highest performing specialized Java processor for the embedded environment [6]. This allows us to gauge our enhancements in comparison to a competitive baseline model.

5.1.1 Contributions

We propose a microarchitectural enhancement that combines multiple stack based instructions into a single register based instruction and caches it to exploit reuse. Synthesis of the Verilog source code model of picoJava-II provided by Sun Microsystems in their Community Source Licensing effort [1], revealed that instruction folding performed by picoJava-II (in the decode stage) falls in the critical path. We propose to remove instruction folding from the decode stage, using a fill unit [79, 46] and a decoded bytecode cache (DB-Cache). By moving pattern checking and folding logic from the decoder to the fill unit, we effectively remove it from the critical path. The fill unit dynamically groups decoded bytecodes into a DB-Cache line, which can be fetched and executed in parallel when the same bytecodes are executed next time. The fill unit and the decoded bytecode cache also exploit bytecode reuse in Java applications. The JVM uses a variable length instruction set and the DB-cache eliminates the need to dynamically identify instruction boundaries.

Another contribution is a dynamic optimization performed in hardware called *stack disambiguation*, which exposes more parallelism in the bytecode stream. Use of the Java stack as an operand scratch pad for computations and as a local variable storage area results in enforcement of false dependencies while exploiting parallelism. Stack disambiguation uses hardware logic to distinguish between the operand stack (at the top of the stack) and the local

[1] In late 1998, Sun Microsystems instituted community source licensing for picoJava-II enabling tool vendors, chip developers, universities and research organizations to experiment with the Java processor RTL source code in verilog. The source code distributed is the same source RTL code Sun Microsystems used for their picoJava-II chip [1].

variable area (at the bottom of the stack). Such a logical distinction between references to these distinct areas in the stack exposes more parallelism, which can be exploited by simple in-order multiple issue engines.

5.1.2 Related Work

The fill unit and decoded instruction cache were hardware assists proposed by Patt et. al. as part of the HPS design philosophy [86, 84]. They describe the fill unit, which compacts instructions generated from a serial instruction stream into a decoded instruction cache. The use of these hardware assists on an HPS version of the DEC VAX processor was seen to result in significant performance improvements [84, 85]. The idea of a fill unit was revisited by Franklin and Smotherman in 1994, to dynamically group RISC instructions into a VLIW like instructions, and showed up to two times improvements over a single issue processor [46]. In a later paper, Smotherman and Franklin showed that collecting decoded X86 instructions using a fill unit and storing them in a decoded instruction cache can improve the decode rate as compared to a P6-like decoding structure [47]. They also presented a register allocation and renaming scheme using the fill unit approach to alleviate the excessive bandwidth placed on the P6 register renaming hardware. The picoJava-II does not require additional renaming stage or excessive ports to the Stack Cache register file, as it already has multiple ports to facilitate instruction folding optimizations.

Fill unit based optimizations for trace caches [24] and trace cache based preprocessing [55] improves performance by transforming instructions within a trace line. However, trace caches have been proposed for machines which can execute up to 8 or 16 instructions in parallel and hence store multiple basic blocks in a trace cache line. In marked contrast, we store only one basic block of bytecodes in the DB-Cache, as we are limited by the hardware resources which we can add in an embedded environment. We perform *instruction folding* within a DB-Cache line, instead of across basic blocks to reduce complexity. As an example, the designers of picoJava-II decided against adding a sophisticated dynamic branch predictor to the picoJava-II and instead opted for a static branch predictor. Investing the hardware resources for dynamic predictor was considered an overkill in terms of area and power. For high end server applications, dynamic branch predictors and aggressive ILP techniques are relevant. Object oriented architectural support in hardware is another approach taken to improve performance in Java processors, which tries to provide support for manipulating objects, specialized caches and other such optimizations [76, 78].

5.2 Background and Motivation

To motivate the research presented in this chapter, we perform an analysis of the decode and folding stages of the picoJava-II processor pipeline in detail. The picoJava-II is a stack based processor, since it is a silicon version of the stack-based Java Virtual Machine (JVM). We show that the decode stage is the longest stage in the pipeline (using critical path analysis), and the delay can be reduced by removing the folding logic from the decode stage.

5.2.1 Decoding instructions in picoJava-II

The picoJava-II core illustrated in Figure 5.1 consists of an integer execution unit, a stack cache, instruction folding hardware and optional instruction and data caches [71].

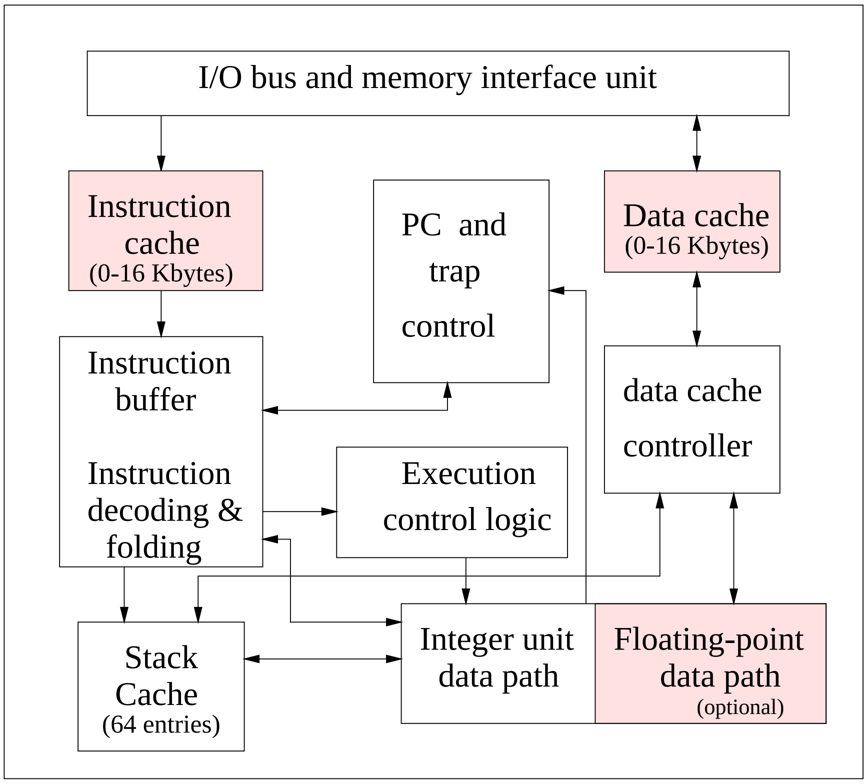

Figure 5.1: Block diagram of the picoJava-II microprocessor core. The caches and the floating point unit are optional.

The instruction and data caches can be configured to be between 0 to 16 KBytes. The stack cache is implemented as a 64-entry register file with random access. The picoJava-II core implements the commonly executed bytecodes in hardware and handles the remaining ones using microcode and software traps. The pipeline in the picoJava-II is implemented in six stages similar to a simple RISC pipelines (with modifications to reflect the stack based nature of the bytecode ISA). The core fetches the instructions from the instruction cache into an instruction buffer in the fetch stage. The next stage decodes the instruction and performs folding optimizations. The stack cache is read for operands in the register stage, if necessary. The instruction executes in one or more cycles in the execute stage and goes through the memory access stage if it needs data from the data cache. Results are written back to the stack cache during the final stage in the pipeline.

A 16 byte instruction buffer is used to decouple the instruction cache from the rest of the pipeline. The processor can write a maximum of 8 bytes into the buffer at one time, where as it can read 7 bytes from the buffer at once. The bytecode ISA has variable length instructions, and most bytecodes consist of a 1 byte opcode, followed by 0, 1 or 2 operands. Since the average length of bytecode instructions is approximately 1.8 bytes [78], the processor can potentially read more than one instruction in a cycle. The processor can read up to four instructions, depending on the length of the instructions in the buffer.

5.2.2 Folding optimizations performed in picoJava-II

The design choice of a stack based machine facilitates the advantage of small instruction (bytecode) size, but causes the generation of abundant load and store operations to the stack. For example, to compute an expression `a+b=c`, the sequence of bytecodes generated are:

- iload_a ; load local variable a into TOS (Top Of Stack)
- iload_b ; load local variable b into TOS
- iadd ; Pop top 2 elements and add them. Push result to TOS
- istore_c ; Store the element on TOS to local variable c

Figure 5.2 shows a snapshot of the stack after execution of each bytecode in the example shown above (a, b and c are assumed to be at location 1, 2 and 3 in the local variable area) to compute `a+b=c`. The picoJava-II takes four cycles to compute `a+b=c`, due to the requirement that bytecodes operate only on the elements at the top of the stack. The processor tries to eliminate the unnecessary loads and stores to the stack by utilizing an *instruction folding* technique. This is possible because the stack is implemented as a register file in the picoJava-II with 3 read ports, two write ports and a single cycle access to any entry of the stack. When a pattern like the one shown earlier is detected, the picoJava-II replaces it with a three-operand instruction `add c← a, b` which takes advantage of the single-cycle random access to the stack cache. Figure 5.3 shows how the picoJava-II computes `a+b=c` in a single cycle, by eliminating the unnecessary loads and stores to the stack cache using instruction folding.

The folding optimization is implemented in the picoJava-II as follows:

1. Folding logic is added to the decoder design to detect patterns of instruc-

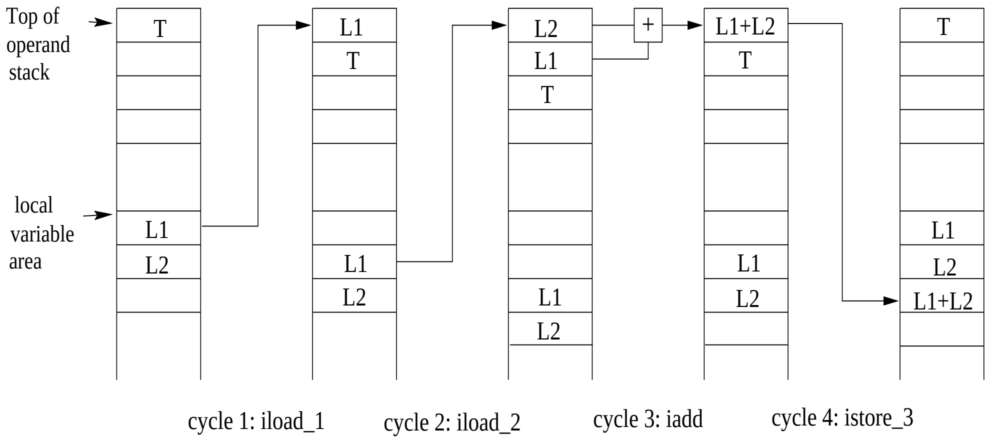

Figure 5.2: Execution of the code sequence iload_1, iload_2, iadd, istore_3 without instruction folding

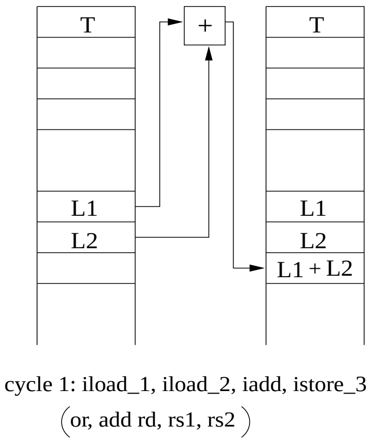

Figure 5.3: Execution of the code sequence iload_1, iload_2, iadd, istore_3 with instruction folding

tions which can be folded together. The pipeline stages for a picoJava-II processor which implements the folding optimization is shown in Figure 5.4. Since there are innumerable folding patterns, only those patterns which occur with a high frequency are checked for using the decoder logic. The most common foldable patterns consist of instructions that are between two and four bytecode instructions.

2. Up to four instructions are decoded in the decoder and checked for foldable patterns consisting of four instructions. If no four instruction pattern is detected, then the instructions are checked for a three instruction pattern. In the event that no three instruction pattern is detected, potential for a two instruction pattern is explored, in the absence of which one instruction is sent to the next pipeline stage. If a pattern is detected, then the instructions are *folded* together and one (RISC style register-based) instruction is constructed for that pattern.

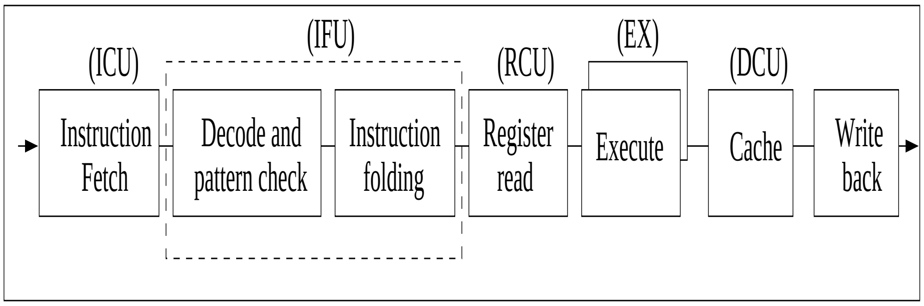

Figure 5.4: Pipeline stages in the picoJava-II processor to implement instruction folding. The stages within the dotted line are responsible for detecting patterns and folding.

5.2.3 Critical path analysis

Folding in the picoJava-II core is performed by the Instruction Folding Unit (IFU) where the instructions are grouped and pre-decoded. The picoJava-II documentation names the decode stage as the IFU, although both decoding and instruction folding is performed in this stage. The IFU classifies instructions into different types and groups them depending on their opcodes. Each of the bytes in the I-Buffer have other information associated with them, namely, a 4-bit length information corresponding to that opcode, a valid bit and a dirty bit. The folding logic examines the top seven bytes of the instruction buffer and determines the number of instructions that can be folded depending on the bytecode, and length information (associated with that bytecode) from the Instruction Cache Unit (ICU).

We analyzed the timing of all the single-cycle units in the picoJava-II core using the Verilog hardware description language model obtained from Sun Microsystems. Based on maximum timing optimizations in Synopsys targeting LSI Logic's 3.3V G10TM-p Cell-Based 0.29 micron ASIC library [9], we synthesized the Verilog code. Table 5.1 shows the results of the synthesis of various stages of the picoJava-II core. The pipeline stage that is the slowest is the IFU (decode+folding) stage.

Figure 5.5 shows the logic elements in the critical path in the Decode pipeline stage (IFU). The length decoder computes the lengths of prospective instructions to be folded using special adders called *index adders, Iadd*. These adders use length information of each byte (l_0 to l_6) and the position of the

Stage/Unit	description of the pipeline stage	delay (ns)
ICU	Instruction cache unit	4.00
IFU (D)	Instruction Folding Unit	12.74
IFU (D')	Modified decode stage (after removing folding)	8.68
RCU	Register cache unit	9.33
EX	Execute stage	2.58
DCU	Data cache unit	5.48

Table 5.1: Critical path delay for the picoJava-II datapath and control units

byte in the I-Buffer. Symbols (L_0 to L_3) represent the cumulative length of one to four instructions. For example, L_2 represents the sum of lengths of the first three instructions out of the four prospective foldable instructions. To compute L_3, L_2 needs to be computed which in turn depends on the earlier cumulative length, elongating the critical path.

The folding decoder consists of an array of folding decode (*FDEC*) blocks, each of which examines two consecutive bytes and generates information about prospective foldable instructions. Two bytes are used by each block due to the fact that the picoJava-II opcodes can either be one byte or two bytes long[2]. Using the accumulated lengths and the *FDEC* block outputs, folding patterns can be identified. Associated folding logic uses these patterns to output the number of foldable instructions and also a shift signal to the I-Buffer to move the contents of the buffer for decoding in the next cycle.

We isolated the folding logic from the decode stage and present the timing analysis of the modified decode stage (D') in Table 5.1. Isolating the

[2]The picoJava-II uses special two byte opcodes for implementation specific extensions to the JVM instruction set [50]

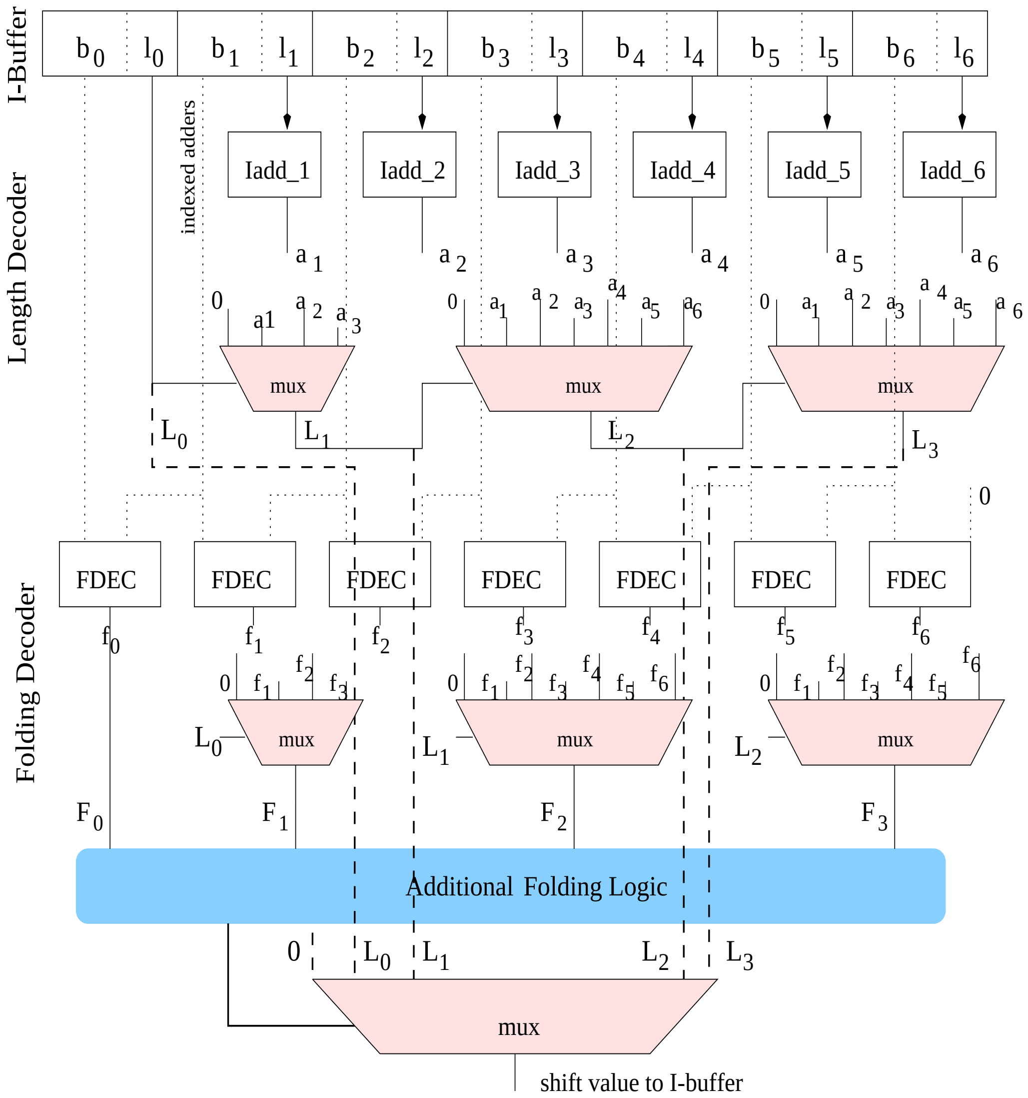

Figure 5.5: Logic along the critical path of the decode stage.This figure shows the logic that contributes to the critical path , in the decode stage (additional logic occurring in parallel is not shown).

folding logic from the critical path brings down the maximum delay from 12.74 ns to 8.68 ns, thus allowing to drive the clock at a faster rate than it would have been possible with folding logic in the critical path. We propose to move the instruction folding logic from the decode stage to the fill unit. The fill unit is outside the critical path of the pipeline, and does not affect the clock as it can take multiple cycles to operate without affecting the performance. After the folding logic is removed from the critical path of processor and added to the fill unit, the register cache unit (RCU) becomes the longest stage in the pipeline (having a delay of 9.33 ns). Therefore, by moving the folding logic to the fill unit we can increase the clock rate of picoJava-II by approximately 25%, which is a significant improvement. Increase in clock frequency will however result in increase in processor power. Embedded applications with power as the primary design constraint may want to make use of only the other proposed microarchitectural enhancements.

5.3 Fill unit and DB-Cache operation

Figure 5.6 shows the fill unit approach that we propose in this chapter. The fill unit collects decoded bytecodes and stores them into the lines of a decoded bytecode cache (DB-Cache). During instruction access, if the instruction fetch address matches an entry in the DB-Cache, the entry in the DB-Cache is given priority over the normal instruction cache and decoding path, and decoded bytecodes are supplied directly to the operand fetch stage. It should be noted that by storing the decoded bytecodes in the DB-Cache, there is no need to repeat fetching of instructions from the normal instruction cache nor finding the instruction boundaries[3]. However, operand fetch must still occur before the instructions can be fed to the processor core for execution. The advantages of the fill unit are that it can collect more than two decoded instructions into the same DB-Cache line and pass to the operand fetch stage in one cycle. Thereby, we increase the instruction decode bandwidth without increasing the decoder width, the fetch bandwidth and the instruction buffer size.

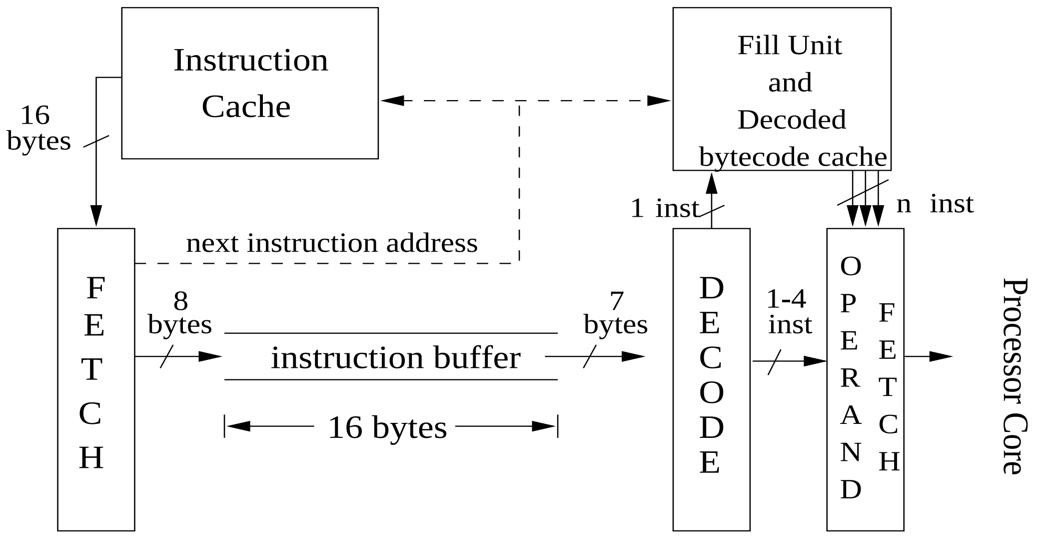

Figure 5.6: Increasing decode bandwidth using a fill unit and DB-Cache. The processor core is fed decoded instructions from the normal decode path, or from the DB-Cache (when there is hit in the DB-Cache).

Since a variable number of decoded bytecodes form a DB-Cache line, each line should also include a next-address field and supply it to the branch unit. Bytecodes are filled into the DB-Cache line only if they represent more than one instruction, as it is more efficient to fetch single instructions from

[3]The picoJava-II has variable length instructions, and instruction boundaries are calculated in the IFU stage while decoding the bytecodes.

the instruction cache. The DB-Cache line could potentially include as many bytecodes as possible, but we restrict it to five decoded bytecodes for this study. We place an additional restriction that only bytecodes of length three bytes and below are stored in the DB-Cache. We are able to capture more than 99% of the bytecodes executed (even with this restriction), as most of the longer length bytecodes are executed very infrequently. Based on these constraints we would require 15 bytes (5 times 3) in each DB-Cache line to store up to five decoded bytecodes.

Data dependencies will not prevent filling of the decoded bytecodes, but will be resolved by the processor core similar to instructions fetched along the normal path. Control dependencies, such as branches will stop the filling of a DB-Cache line in our design. We could use a more complex design where we fill past branches (to store multiple basic blocks), however, that would prove to be too expensive in terms of resources (additional memory and branch prediction logic) for the picoJava-II processor. The next-address field in each DB-Cache line is used to provide correct instruction sequencing. If there is no branch or control-changing instruction in a DB-Cache line, the next-address field is set to the first instruction that was not filled into the current line. A branch is represented in the DB-Cache line by two fields - a condition and a branch address. When the last instruction to be filled is a conditional branch, the next-address field serves as the branch-untaken address, and the branch address field serves as the branch taken address. This is to facilitate pre-fetching of lines from the DB-Cache by using the branch predictor to predict the outcome of the branch. The predictor output is used to generate the next instruction address and sent to the instruction cache and the DB-Cache. The total DB-Cache line size for this design will be 23 bytes (15 bytes for decoded bytecodes and 8 bytes to store next-address and the branch address).

5.3.1 Performing folding operations

In the previous section we saw how the DB-Cache can be used to increase the decode bandwidth for the picoJava. We now investigate some optimizations which can be performed by the fill unit while filling DB-Cache lines. One such optimization is *instruction folding.* As explained in the Section 5.2.3, implementing the folding optimizations in hardware requires adding additional logic to the decode stage. This could have a serious impact, especially if we want to detect many patterns, and patterns consist of large instructions. Since the folding logic falls in the critical path of the processor, it has to be highly optimized and kept to a bare minimum (in terms of functionality). Instead, we can use the fill unit to perform the folding operations while it is filling the decoded instructions into the DB-Cache, and this in turn has its own advantages. We explain how the fill unit performs the folding operations using Figure 5.7.

The fill unit collects instructions as described earlier and checks for instructions that can be stored in a DB-Cache line. We incorporate the pattern check and folding logic in the fill unit so that it checks for patterns in the bytecodes that it collects. If it detects a pattern, then it can store the synthesized

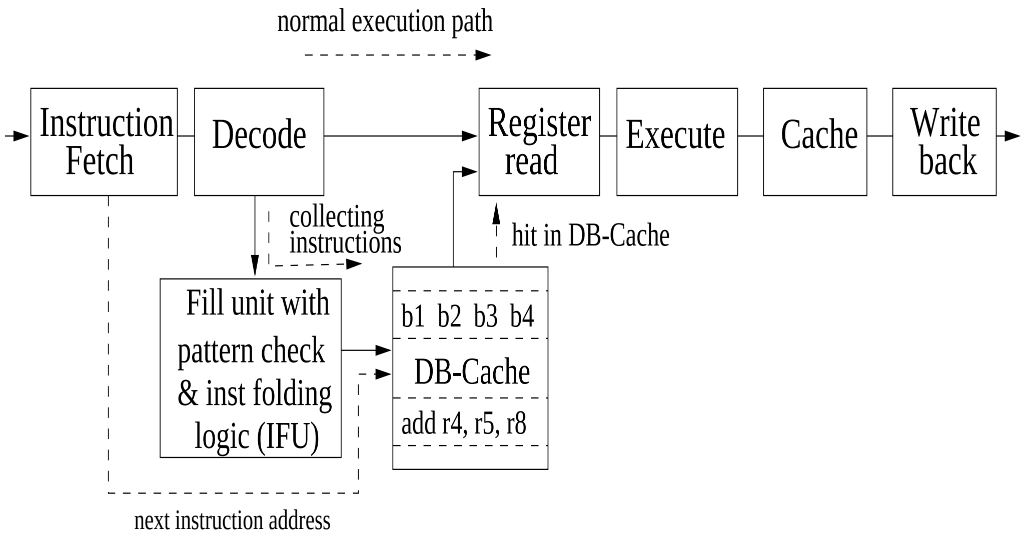

Figure 5.7: Storing basic blocks and *foldable patterns* of bytecodes in the DB-Cache. The DB-Cache will store basic blocks of bytecodes as well as RISC-style instructions (folded bytecodes).

(register-based) instruction and substitute it for the pattern of bytecodes. This is then stored in the DB-Cache instead of the bytecodes themselves. In Figure 5.7, [b1, b2, b3, b4] is a basic block of bytecodes which is collected and filled into the DB-Cache. The bytecodes are stored without modification (folding) if they do not form a foldable pattern. During the execution of the program if the fill unit came across a 4 bytecode sequence which matched a foldable pattern, the fill unit fills the DB-Cache line with the synthesized RISC-style instruction. A DB-Cache line could contain a mix of both a synthesized instruction and bytecodes as long as they do not exceed the DB-Cache line size (and are part of the same basic block). The advantages of incorporating the pattern checking and instruction folding in the fill unit are as follows:

1. By moving the pattern checking and folding logic from the decoder to

the fill unit, we are effectively removing it from the critical path of the decode stage. Now that the folding logic is not in the critical path we can implement other patterns (less frequent) that were not included because of cost and timing considerations.

2. The decode width of the processor need not be increased to identify patterns which are wider than the decode width. Since the picoJava-II has a decode width of 7 bytes(1-4 bytecodes depending on their length), it can identify patterns of 4 bytecode instructions. If it has to identify instruction patterns of 5 instructions, the decode width will have to be increased from 4 to 5 (by increasing the I-Buffer and decode logic). If we move the pattern checking and folding logic to the fill unit, we still need to only have a decode bandwidth of 4, as the pattern check is done by the fill unit when collecting the decoded instructions (the DB-Cache line holds up to 5 instructions). Along the same lines, the picoJava-I core [50] has a decode width of two, and can fold only up to 2 bytecodes. If we add a fill unit and DB-Cache (similar to the one proposed in this chapter), and move the folding logic to the fill unit we will be able to detect patterns of 3 and 4 in picoJava-I without increasing the decode width.

3. In the picoJava-II we have to check for patterns and perform folding for a sequence of bytecodes, no matter how many times it is executed. However, using the fill unit we replace the pattern with the synthesized RISC-style instruction and store it in the DB-Cache line, and then we can

issue the synthesized instruction when the pattern is repeated (assuming we have a DB-Cache hit). In the picoJava-II (without the DB-Cache) the bytecodes would have to go through the decode stage and perform the folding operation again, before it can be issued thereby taking more cycles.

5.4 Effectiveness of the fill unit and DB-Cache

The effectiveness of the proposed techniques are evaluated using a picoJava-II simulator that accepts bytecode traces extracted using a tracing JVM [63]. The simulator models the picoJava-II pipeline on a cycle by cycle basis using instruction latencies[4] of each bytecode. Single cycle instruction and data caches with 100% hit ratio are assumed. The branch predictor used in the picoJava-II is a static predictor, and has a penalty of 3 cycles for mis-predicted branches (not taken branches).

5.4.1 Instruction decode rate

Figure 5.8 shows the increase in decode rate which can be achieved using the fill unit and the DB-Cache. The decode rate increases as we increase the number of entries in the DB-Cache from 64 to 16K. DB-Cache with more than 1K entries are shown only for comparison purposes, since it would not be feasible to have such a big cache structure in the picoJava-II core (due to its power

[4]The instruction latencies for the bytecodes are obtained from [72]. We assume a 30 cycle latency for software traps. These traps account for less than 0.1% of the total instructions.

and area impact).

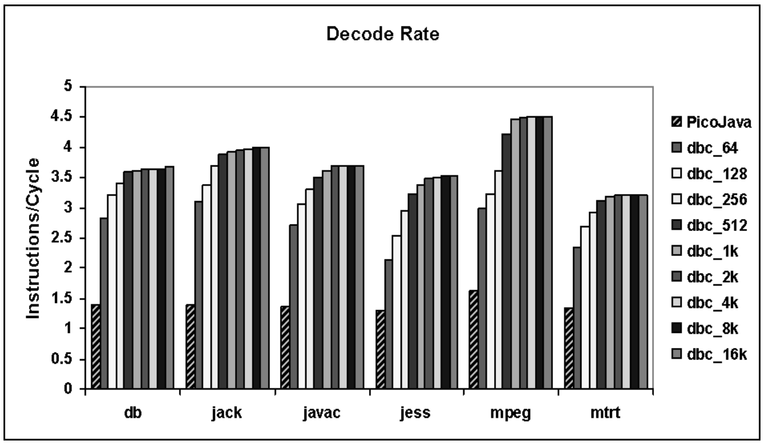

Figure 5.8: Decode rate for different DB-Cache sizes

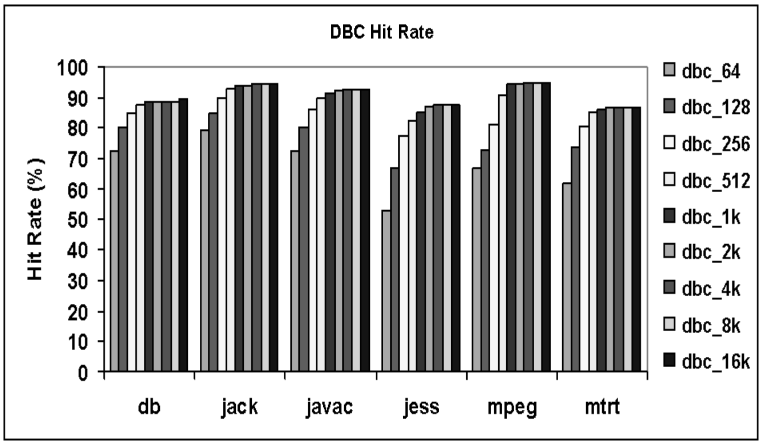

Figure 5.9: Hit rate for different DB-Cache sizes

The picoJava-II processor could potentially decode up to four instructions (due to folding), and we see a decode rate of 1.4 to 1.7 across the different benchmarks (first bar in Figure 5.8). We can achieve a two-fold increase in decode rate even with a DB-Cache of only 64 entries. The decode rate increases to up to three-fold for a 1K entry DB-Cache. The increase in decode rate corresponds to the hit rate that is seen for the different DB-Cache configurations (Figure 5.9).

5.4.2 Folding efficiency and performance

In this section, we look at the folding coverage and percentage of instructions that are eliminated when moving the folding logic to the fill unit. As described in the previous section, the number of bytecode patterns detected would depend on the hit rate of the DB-Cache, and certain other factors which we describe in this section.

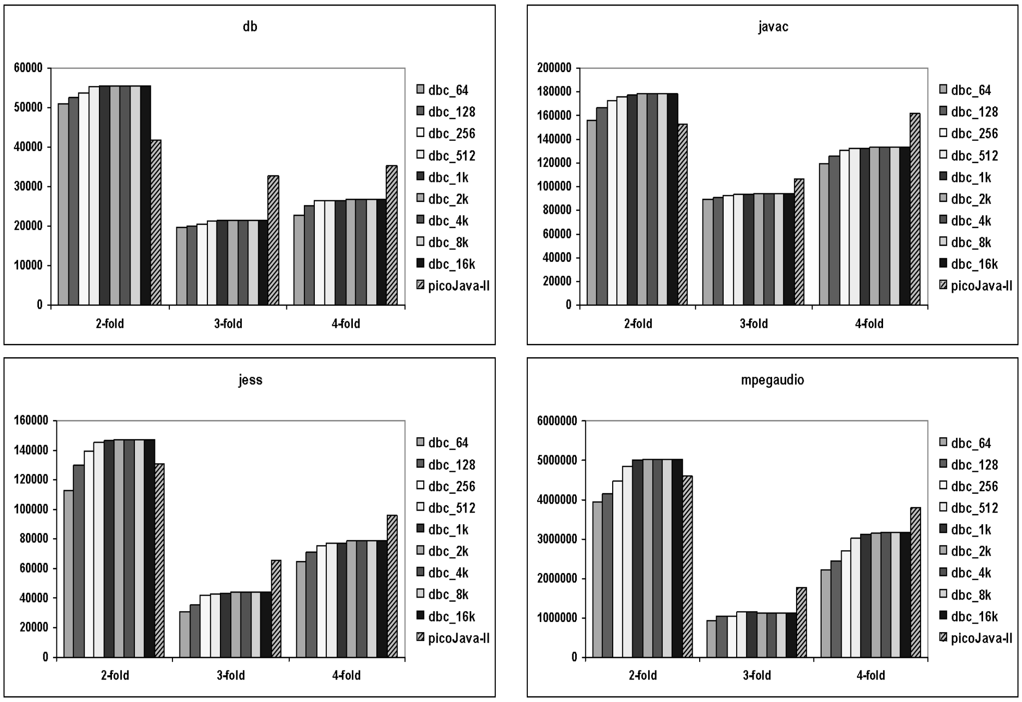

Figure 5.10: Folding patterns detected in picoJava-II (at decode stage) and when using the fill unit and DB-Cache.

Figure 5.10 shows the absolute number of patterns detected for each benchmark in the picoJava-II processor with and without the fill unit and DB-Cache. The total number of folded two instruction patterns (2-fold), three instruction patterns (3-fold) and four instruction patterns (4-fold) are shown for each benchmark. The last bar in the graph (named as picoJava-II) shows the number of patterns detected when the pattern check is done at the decode stage of the picoJava-II processor. The number of three instruction and four instruction patterns detected in the fill unit is seen to be lower compared to the normal picoJava-II processor. This is mainly due to two reasons; (i) We fail to detect a pattern the first time the fill unit encounters it, since it follows the normal decode and execute path, which does not have pattern checking and folding logic, and (ii) some of the larger patterns, especially four instruction patterns get stored in two lines due to overflow. For example, if a 4 instruction pattern starts at the middle of a DB-Cache line, and continues into the next cache line (because the pattern starts at the middle of a basic block stored in that line) we would fail to detect it. We refer to this as *pattern fragmentation*. Because of pattern fragmentation, a four instruction pattern might avoid being detected when using the fill unit. However, if the split pattern forms a smaller instruction pattern (of two or three instructions) which is stored in one DB-Cache line, the pattern checker in the fill unit will detect it. Thus, there is a potential that the total number of two instruction patterns that are detected will be more when we do the pattern checking in the fill unit. It can be seen in Figure 5.10 that this is the case, and the two instruction patterns (2-

fold) that are detected using the fill unit exceed that detected by the normal picoJava-II processor (as these 2 instruction patterns were part of a longer 3 or 4 instruction pattern, which was detected in the picoJava, but went undetected in the fill unit).

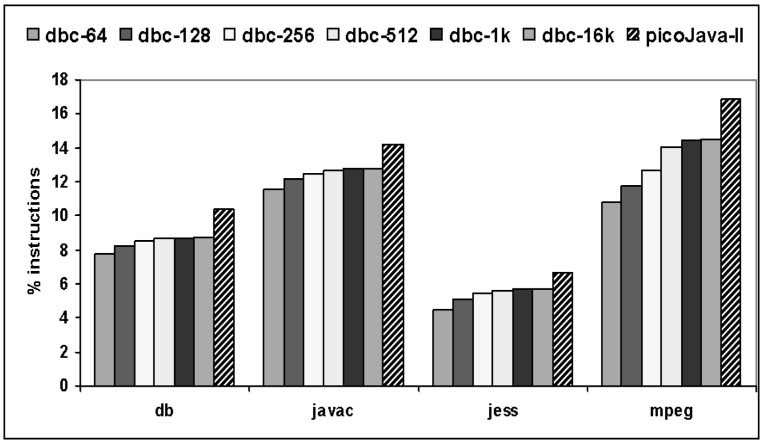

Figure 5.11: Percentage of total dynamic instructions that are eliminated (folded) using instruction folding

Figure 5.11 shows the percentage of total dynamic instructions which were folded for four of the SpecJVM98 benchmarks, based on the instruction patterns (2-fold, 3-Fold and 4-Fold) that were detected in the bytecode execution stream. The percentage of instructions eliminated when folding is done at the decode stage is 2% higher compared to using a DB-Cache (to hold the folded patterns) of 64 entries for *db*, *javac* and *jess*. This percentage difference increases to 6% for *mpeg*, but goes down to almost 2% when using a DB-Cache entry of 1k entries. It can be inferred from the graph, that using a fill unit to capture and store foldable instruction patterns in the DB-Cache allows to eliminate almost the same percentage of instructions as when doing the folding at the decode stage. Failing to detecting a pattern the first time it is encountered, *pattern fragmentation* and misses in the DB-Cache cause the percentage of instructions to be slightly lower than what one would observe when folding is done at the decode stage.

5.5 In-order multiple instruction issue

One straightforward technique to improve performance is to exploit parallelism in the instruction stream using ILP techniques. However, obtaining increased performance in a resource-constrained embedded processor involves detecting and executing multiple instructions in a cost-effective manner. The previous sections illustrated how a fill unit and DB-Cache can achieve up to a three-fold increase in the decode rate. In this section, we investigate the performance improvement achieved by executing multiple instructions in conjunction with the fill unit and DB-Cache which provide an abundant supply of instructions. While we want to exploit instruction level parallelism, we would like to keep the complexity manageable for embedded environments and hence restrict the exploration to in-order execution as opposed to using aggressive ILP techniques which require substantially higher amount of resources.

5.5.1 Effectiveness of in-order multiple instruction issue

To detect the level of exploitable concurrency in the bytecode stream, we relax the resource constraints on number of execution units and estimate the optimum number of execution units. When the execute stage is fed a basic block of instructions from the DB-Cache, it could potentially execute all of them in parallel assuming there are no dependencies and resource constraints. If any instruction in the basic block writes to the stack, then all following instructions which read from the stack will have a dependency on that instruction (and will be executed in the next cycle). We simulate the picoJava-II core with the above constraints for the different benchmarks to find the potential execution rate assuming a DB-Cache of 128 entries. It should be noted that no exploration beyond basic blocks is done and what is revealed here is by no means the bound on available parallelism in the whole application.

	Instructions Executed in Parallel				
benchmark	1	2	3	4	5
db	90.12	9.88	0.00	0.00	0.00
javac	93.41	6.28	0.17	0.00	0.00
jess	92.05	7.95	0.00	0.00	0.00
mpeg	94.70	5.30	0.00	0.00	0.00
mtrt	91.22	8.78	0.00	0.00	0.00

Table 5.2: Percentage of instructions executed in parallel when using a DB-Cache of 128 entries

Table 5.2 shows the percentage of instructions that can be executed in parallel for the picoJava-II with a fill unit and DB-Cache (of 128 entries). It is seen that 90-94% of the instructions are executed without being paired with other instructions. *Javac* is one benchmark in which we see a negligible small percent of the instructions being executed at a rate of 3 instructions per cycle (0.17%). We never see 4 or 5 bytecodes being executed in one cycle, because of the dependencies to the stack. This implies that in the limited window we are investigating, most of the instructions have a dependency to the stack, and is limiting the execution width even if we are able to decode up to 5 instructions in one cycle. Since there is one unique stack, an operation which expects an operand from the stack must wait and depend on the previous load or written back data to be stored into the stack. This causes a severe limitation to the instruction level parallelism that can be exploited. Therefore, it we will be not be possible to gain much performance by adding resources to execute more than 2 instructions in one cycle, without alleviating the constraints imposed by the stack.

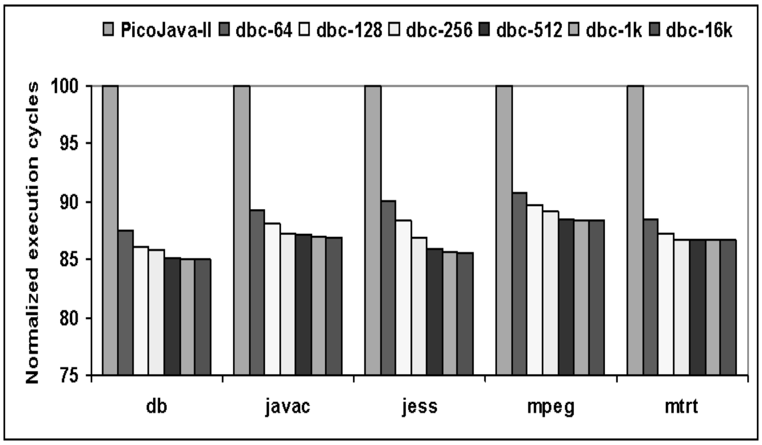

Figure 5.12: Performance improvement when adding a fill unit, DB-Cache (64-16K entries) and instruction execute width of two to a picoJava-II processor.

Figure 5.12 shows the relative performance of a dual issue architecture with the proposed fill unit and a few different DB-Cache configurations. This architecture can execute pairs of bytecodes if they have no dependencies. If one bytecode has a longer latency than than its pair, the pipeline stalls till the longer latency instruction completes. A DB-Cache of 64 entries yields a performance improvement of 10% to 13%, but performance does not improve beyond 15% for larger DB-Caches. Even though this is not a very significant improvement in performance, it should be kept in mind that this is in addition to the clock speed improvements made possible by the fill unit and the DB-Cache. Performance improvement is limited primarily by stack dependencies. In the next section, we show how we can alleviate some of this problem by using an optimization called the *stack disambiguation.*

5.5.2 Using stack disambiguation to exploit concurrency

The Java stack is used to hold the operand stack, local variable area and frame data for each method. The operand stack is used as a scratch pad to perform computations and store values. All operations on the operand stack take place at the top of the stack and the OPTOP (top of operand stack) register is used to manipulate the operand stack. The local variable area consists of values for commonly used local variables for a particular method, and occupies the bottom entries of the Java stack. A register VARS is used in the picoJava-II to mark the beginning of the local variable area. The sizes of the operand stack and local variable area varies depending on the requirements of each method. Any data access generated as an offset from the OPTOP or VARS register in the picoJava, is recognized as a stack access and all other data access go the data cache. If we use additional logic to distinguish between stack access generated as an offset to the OPTOP and VARS register, we can schedule more instructions in parallel since the operand stack is the true stack, and the local variable area is a storage area. We refer to this optimization to distinguish between operand stack and local variable access as *stack disambiguation*, since it logically separates the stack into the operand stack and local variable area.

	Instructions executed in Parallel				
benchmark	1	2	3	4	5
db	78.76	21.06	0.17	0.00	0.00
javac	81.25	18.17	0.57	0.00	0.00
jess	81.06	18.20	0.74	0.00	0.00
mpeg	87.32	12.47	0.21	0.00	0.00
mtrt	86.44	11.56	2.00	0.00	0.00

Table 5.3: Percentage of instructions executed in parallel when using *stack disambiguation* with a 128 entry DB-Cache)

Using additional logic to check the type of access, could affect the cycle time if it falls in the critical path of the data cache access. Instead we could use an additional bit along with each instruction stored in the DB-Cache, to mark if it is an access to the operand stack or the local variable area[5]. The percentage of instructions executed in parallel, when using *stack disambiguation* is given in Table 5.3. As expected, we see an improvement in the execution rate, and more instructions being executed in parallel, when compared to the data in Table 5.2 where we do not distinguish between stack and local variable access. The percentage of instructions that are executed with a width of three increases, and we see 0.17% to 2% of the instructions in this category. In Table 5.2 only *javac* was seen to have a small percentage of instructions which were executed with a rate of 3 instructions per cycle. Similarly, we see 2 to 3

[5]This would add 5 bits to each DB-Cache line for the configuration used in this study

times improvement in the percent of instructions that are executed in pairs.

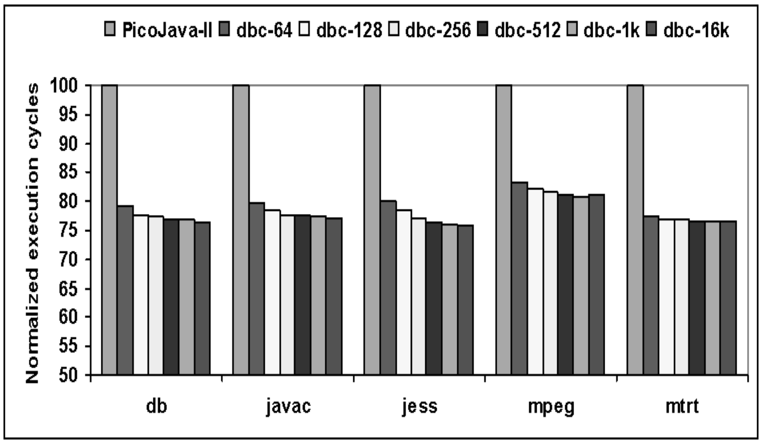

Figure 5.13: Relative performance of picoJava-II using the fill unit, DB-Cache (64-16K entries), execution width of two and *stack disambiguation*

The performance of the picoJava-II processor with the fill unit configuration and the *stack disambiguation* is plotted relative to the performance seen for the original picoJava-II in Figure 5.13. Performance improvements range from 17% to 24% for the different DB-Cache configurations. *Stack disambiguation* alone is seen to contribute approximately 10% over a naive stack access implementation.

5.6 Summary

Due to their low cost and low power requirements, achieving high performance in Java processors necessitates a balance of complexity and performance. In this dissertation we extended the concept of fill unit for Java bytecode engines, and explored the effectiveness of fill unit based optimizations. We show that a fill unit coupled with a relatively small decoded bytecode cache (64 entries) improves performance, showing that cost-effective techniques exist to exploit limited ILP for the low-end embedded Java processors. The major contributions are summarized below:

- Performing instruction folding using the fill unit and DB-Cache allows to increase clock frequency by approximately 25%. The picoJava-II core performs instruction folding in the decode stage. Based on maximum timing optimizations in Synopsys targeting LSI Logic's 3.3V G10TM-p Cell-Based 0.29 micron ASIC library [9], synthesizing the Verilog code for the picoJava-II core showed that instruction folding elongates the critical path.

- Adding a fill unit and DB-Cache improves decode bandwidth by 2x to 3x. This result is obtained from simulation studies using the SpecJVM98 benchmarks on a picoJava-II simulator with fill unit and DB-Cache. This improvement in decode width is obtained without increasing the baseline decoder width, fetch width and the instruction buffer size.

- Instruction folding in the fill unit may not detect all foldable patterns, however the loss in performance is negligible. Failing to detect a pattern during its first occurrence, detecting only a part of the pattern (*pattern fragmentation*) and misses in the DB-Cache reduce the overall number of dynamic instructions folded by approximately 2%.

- An in-order dual issue execution core with a 64-entry DB-Cache improves performance by 10% to 14%. Increasing the number of functional units to more than two is not beneficial due to stack dependencies. It is seen that 90% to 95% of the instructions cannot be paired and issued in parallel due to dependencies on the stack.

- Stack disambiguation increases pairing of instructions by eliminating false dependencies. Logically separating the stack accesses to the operand stack and the local variable area allows more instructions to be executed in parallel. Dependencies to the stack is *the* limiting factor for exploiting ILP in stack-based architectures. Using our technique of *stack disambiguation* allows for an additional improvement of 10% over the fill unit and DB-Cache configuration, leading to an overall improvement of 17% to 24% in execution cycles over the picoJava-II processor. The performance improvement has been quantified in execution cycles, and the improvement will be higher if we consider execution time, as our optimizations are orthogonal to the 25% improvement to the clock frequency.

It may be noted that we have not investigated any out-of-order ILP techniques in this chapter. Incorporating fill unit and the DB-Cache into the picoJava-II core was seen to improve performance in many different aspects. Implementing aggressive ILP techniques to improve performance in embedded processors has to be done judicially, keeping in mind the power and area impact of the optimizations.

Chapter 6

Conclusions

Java is an exciting new object-oriented technology and has led to interesting ways to develop code in embedded, mobile and general purpose community. The popularity of Java is fueled mainly by its platform independence, object technology and security. A growing concern is the performance penalty associated with Java applications, and forms the focus of this dissertation.

6.1 Summary

In this dissertation, we investigated the architectural implications of JVM implementations with real applications, for enabling the development of efficient Java runtime systems on a wide range of platforms and execution environment. The bottlenecks during interpretation, Just-In-Time (JIT) compilation, and hardware realization were analyzed.

The primary bottleneck observed in execution of Java applications on general-purpose processors was the overhead of translation. In interpretation, it was the overhead of software fetch and decode. In JITs, it was the overhead of performing compilation and compiler optimizations at run time and the extra memory requirements of the JIT. The overheads are more prominent for short-running, client-like workloads, where the execution time is spread over a large number of methods.

The primary contribution of this dissertation is the Hard-Int architecture which dynamically translates bytecodes to native machine instructions using a hardware translation unit and a translated code buffer. The hardware translation unit fetches the bytecodes from the instruction cache and predecodes them before performing the translation using a microcoded ROM. We compared the performance of SPECjvm98 benchmarks when executed using software translation (interpreter and JIT) to hardware translation using the Hard-Int architecture.

Hardware translation was seen to result in an average speedup of 543% over the JDK 1.2 interpreter on a 4-way processor. The Hard-Int architecture performs consistently better than a JIT compiler for client workloads, showing an average speedup of 259% over the JDK 1.2 JIT compiler on a 4-way machine. The average is significantly affected by *mpeg*, which is the only program that shows a performance degradation. Excluding *mpeg*, the average speedup is 376% over the JIT compiler on a 4-way machine.

We then found that the stack-based nature of the JVM was limiting the available parallelism in a Java processor. To address this issue, we proposed microarchitectural techniques, (using the PicoJava-II core as a reference model) to expose more parallelism in the bytecode stream. We showed that a fill unit coupled with a relatively small decoded bytecode cache improves performance, proving that cost-effective techniques exist to exploit limited ILP for the low-end embedded Java processors.

Performing instruction folding using the fill unit and DB-Cache allowed to increase the clock frequency by approximately 25%. In addition, an in-order dual issue execution core with a 64-entry DB-Cache improved performance by 10% to 14% over the PicoJava-II core. It was seen that 90% to 95% of the instructions cannot be paired and issued in parallel due to dependencies on the stack. Logically separating the stack accesses to the operand stack and the local variable area allowed more instructions to be executed in parallel. We presented a dynamic hardware optimization called *stack disambiguation* which provided an additional improvement of 10% over the fill unit and DB-Cache configuration, leading to an overall improvement of 17% to 24% in execution cycles over the picoJava-II processor.

In addition to the aforementioned microarchitectural enhancements presented in chapter 4 and 5, the detailed workload characterization presented in chapter 3 is a significant contribution of this dissertation. Identifying the bottlenecks in Java execution and quantifying individual sources of performance loss and their impact is a crucial step in optimizing Java performance.

6.2 Concluding Remarks

As certain workloads become dominant general purpose computer architectures have added hardware support to execute them efficiently. This was the case with numerical coprocessors being integrated into the processor core as more transistors could fit onto a single die. MMX technology was introduced when media workloads became a dominant part of the desktop workloads. Java technology is emerging as a force in the software industry and Java workloads are becoming an important part of general processor workloads. We show that current execution modes of Java using software translation suffers from poor performance compared to compiled native code. Providing hardware support in general purpose processors is one way to bridge the existing performance gap between the execution of Java applications and natively compiled applications.

Java execution can be improved by providing microarchitectural support which target specific features of Java workloads such as polymorphic calls, object allocation, garbage collection, multithreading and virtual method invocation. Vijay et. al. [77, 78, 76] and Sylvia et. al. [25] look at some of these issues in detail. Solutions proposed to the aforementioned bottlenecks allow for incremental improvements in Java performance. However, we found that the most critical bottleneck in current execution modes of Java is the translation overhead. This dissertation shows that providing hardware support for Java bytecode translation improves performance of Java execution significantly.

A dedicated Java processor can also attain significant performance gain, if it efficiently addresses the limitations imposed due to the stack-based nature of the bytecodes. Even if the low ILP issues are successfully addressed, software compatibility and efficient execution of non-bytecode code would remain a major obstacle. Legacy software and compatibility issues were major reasons for the commercial failure of language specific processors in the past. In this dissertation we present a hardware solution which is more efficient than software approaches, is compatible with any general purpose architecture and furthermore does not deteriorate the performance of native (non-Java) code on that processor.

Bibliography

[1] Community Source Licensing for picoJava Technology. http://www.sun.com/microelectronics/communitysource-/picojava/.

[2] IBM, Jikes Project. http://www10.software.ibm.com/developerworks/opensource/jikes.

[3] Kaffe Virtual Machine. http://www.transvirtual.com.

[4] Overview of Java platform product family . http://www.javasoft.com/products/OV_jdkProduct.html.

[5] Patriot Scientific Corporation, the PSC1000 Processor. http://www.ptsc.com/PSC1000.

[6] SPEC JVM 98 Results. http://www.spec.org/osg/jvm98/results/jvm98.html.

[7] SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/.

[8] Tower Technology Corporation, Austin, TX. http://www.towerj.com.

[9] Synopsys Online Documentation, *Guidelines and Practices for Synthesis* v.1997-08.

[10] A. Barisone, F. Bellotti, R. Berta, and A. De Gloria . Instruction Level Characterization of Java Virtual Machine Workload. In *Digest of Workshop on Workload Characterization (WWC-99)*, 1999.

[11] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parakh, and J. M. Stichnoth. Fast effective code generation in a just-in-time Java compiler. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 280–290, June 1998.

[12] G. Aigner and Urs Holzle. Eliminating Virtual Function Calls in C++ Programs. In *ECOOP '96 Conference Proceedings*, pages 142–166, July 1996.

[13] Erik Altman and Kemal Ebcioglu. Daisy: Dynamic compilation for 100% architectural compatability. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[14] Todd M. Austin and Guri S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, May 1992.

[15] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization in Java. In *Proceedings of ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, June 1998.

[16] Per Bothner. A Gcc-based Java Implementation . In *Proceedings of IEEE Compcon'97*, 1997.

[17] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs . *Journal of Programming Languages*, 2(4), 1994.

[18] Yuan Chou and John Paul Shen. Instruction Path Coprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 270–281, June 2000.

[19] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI TR-93-12, Sun Microsystems Inc, 1993.

[20] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time . *IEEE Micro*, 17(3):36–43, May-June 1997.

[21] D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

[22] DashO, preEmptive Solutions Inc, http://www.preemptive.com/.

[23] S. Deickmann and U. Holzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object Oriented Programming*, July 1999.

[24] D.H. Friendly, S.J. Patel and Y.N. Patt. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)*, pages 173–181, 1998.

[25] Sylvia Dieckmann and Urs Holzle. A Case for Using Active Memory to Support Garbage Collection. In *Proceedings of the First Workshop on Hardware Support for Objects and Microarchitectures in Java*, pages 1–5, October 1999.

[26] Karel Driesen and Urs Holzle. The Direct Cost of Virtual Function Calls in C++. In *Procedings of OOPSLA-1996, San Jose, CA*, October 1996.

[27] Karel Driesen and Urs Holzle. Accurate Indirect Branch Prediction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 167–178, June 1998.

[28] Karel Driesen and Urs Holzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 249–258, 1999.

[29] E. Debaere and J. Campenhout. *Interpretation and Instruction Path Coprocessing*. MIT Press, 1990.

[30] D. Griswold. The Java HotSpot Virtual Machine Architecture , March 1998. Sun Microsystems Whitepaper.

[31] Guava - High performance Environment for Running Java Programs, Softway Pty Ltd., 1996.

[32] John L Hennesy and David A Patterson. *Computer Architecture, A Quanitative Approach*. Morgan Kauffman, 1996.

[33] Urs Holzle and David Ungar. Do object-oriented languages need special hardware support? In *ECOOP-95 Conference Proceedings*, August 1995.

[34] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu. Optimizing NET compilers for improved Java performance . *IEEE Computer*, pages 67–75, June 1997.

[35] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu. A study of the cache and branch performance issues with running Java on current hardware platforms. In *Proceedings of the IEEE Compcon '97*, pages 211–216, 1997.

[36] C. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 90–97, 1996.

[37] Ifor Williams, Mario Wolczko and Trevor Hopkins. Dynamic Grouping in an Object Oriented Virtual Memory Hierarchy. In *Proc. ECOOP 87*, pages 79–88, 1987.

[38] Java JIT Compiler. http://www.sun.com/solaris/jit/.

[39] Jeffrey Dean, Greg DeFouw, David Grove, Vass Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of OOPSLA'96*, 1996.

[40] K.Ebcioglu, E.Altman, and E.Hokenek. A Java ILP machine based on fast dynamic compilation. In *In proceedings of MASCOTS'97 - International Workshop on Security and Effeciency Aspects of Java*, 1997.

[41] Alexander Klaiber. The Technology Behind Crusoe Processors, January 2000. Transmeta Corporation White Paper.

[42] A. Krall. Efficient JavaVM just-in-time compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 54–61, 1998.

[43] A. Krall and M. Probst. Monitors and exceptions: How to implement Java efficiently . In *Proceedings of ACM 1998 Workshop on Java for High-Performance Computing*, pages 15–24, 1998.

[44] L.-C. Chang, L.-R. Ton, M.-F. Kao and C.-P. Chung. Stack operations folding in Java processors. *IEE proceedings on Computers and Digital Techniques*, 145(5):333–340, Sept 1998.

[45] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[46] M. Franklin and M. Smotherman. A Fill-Unit Approach to Multiple Instruction Issue. In *Proceedings of Micro-27*, pages 162–171, 1994.

[47] M. Smotherman and M. Franklin. Improving CISC Instruction Decoding Performance Using a Fill Unit . In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, 1995.

[48] M. Tremblay. An Architecture for the New Millenium. In *Proceedings of Hot Chips 11*, August 1999.

[49] S. McFarling. Combining branch predictors. Technical Report WRL Technical Note TN-36, DEC Western Resaech Laboratory, June 1993.

[50] H. McGhan and M. O'Connor. PicoJava: A direct execution engine for Java bytecode . *IEEE Computer*, pages 22–30, October 1998.

[51] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code . In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 1–20, 1997.

[52] T. Newhall and B. Miller. Performance measurement of Interpreted Programs. In *Proceedings of Euro-Par'98 Conference*, September 1998.

[53] M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware . *IEEE Micro*, pages 45–53, March-April 1997.

[54] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications - a way ahead of time (wat) compiler. Technical Report Technical Report, Department of Computer Science, University of Arizona, Tucson, 1997.

[55] Q. Jacobson and J.E. Smith. Instruction pre-processing in trace processors. In *Proceedings of 5th International Symposium on High-Performance Computer Architecture (HPCA-5)*, pages 125–129, 1999.

[56] R. Radhakrishnan, D. Talla and L. John. Allowing for ILP in an Embedded Java Processor. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 294–305, June 2000.

[57] R. Radhakrishnan, J. Rubio, L. John and N. Vijaykrishnan. Execution characteristics of just-in-time compilers. Technical Report TR-990717, The University of Texas at Austin, 1999.
http://www.ece.utexas.edu/projects/ece/lca/ps/tr990717.ps.

[58] R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium. Architectural Issues in Java Runtime Systems. In *Proceedings of 6th International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 387–398, January 2000.

[59] R. Radhakrishnan and L. John. Execution Characteristics of Object Oriented Programs on the UltraSPARC-II. In *Proceedings of the 5th International Conference on High Performance Computing*, December 1998.

[60] R. Radhakrishnan and L. John. Understanding the Branch Performance of Object Oriented Workloads. In *Proceedings of the Workshop on Performance Analysis and Its Impact on Design (PAID98)*, July 1998.

[61] R. Radhakrishnan and L. John. A Decoupled Translate Execute (DTE) Architecture to Improve Performance of Java Execution. In *Proceedings of the First Workshop on Hardware Support for Objects and Microarchitectures for Java*, pages 25–29, October 1999.

[62] R. Radhakrishnan, J. Rubio, and L. John. Characterization of Java applications at the bytecode level and at UltraSPARC-II Machine Code Level. In *Proceedings of International Conference on Computer Design*, October 1999.

[63] Juan Rubio. Characterization of java application at the bytecode level, May 1999. Masters Report, The University of Texas at Austin.

[64] S. Pemberton and M. C. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.

[65] J. Sabarinathan. A study of instruction level parallelism in contemporary computer applications , December 1999. Masters Report, The University of Texas at Austin.

[66] R. Sathe and M. Franklin. Available parallelism with data value prediction. In *Proceedings of the 5th International Conference on High Performance Computing (HiPC-98)*, pages 194–201, April 1998.

[67] Tom Shanley. *Pentium Pro and Pentium II System Architecture*. Addison-Wesley, 1998.

[68] Hank Shiffman. JSTAR: Practical Java Acceleration For Information Appliances. http://www.jeditech.com/jstarwp.html.

[69] Jim E Smith. Decoupled Access/Execute Computer Architecture. In *ACM Transactions on Computer Systems*, pages 289–308, November 1984.

[70] T. B. Steel. A First Version of UNCOL. In *Proceedings of the Western Joint Computer Conference*, pages 371–377, 1961.

[71] Sun Microsystems. *picoJava-II Microarchitecture Guide*, March 1999.

[72] Sun Microsystems. *picoJava-II Programmer's Reference Manual*, March 1999.

[73] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[74] T.H Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J-L. Baer, B. N. Bershad and H. M. Levy. The Structure and Performance of Interpreters. In *Proceedings of ASPLOS VII*, pages 150–159, 1996.

[75] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of SOAR: Smalltalk on a RISC. In *11th Annual Symposium on Computer Architecture*, pages 188–197, June 1984.

[76] N. Vijaykrishnan. *Issues in the Design of a Java Processor Architecture*. PhD thesis, College of Engineering, University of South Florida, Tampa, FL 33620, July 1998.

[77] N. Vijaykrishnan and N. Ranganathan. Tuning Branch Predictors to support Virtual Method Invocation in Java. In *Proceedings of the 5th USENIX Conference of Object-Oriented Technologies and Systems*, pages 217–228, 1999.

[78] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 430–455, July 1998.

[79] W.-M. Hwu and Y.N. Patt . HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality. In *Proc. of 13th Annual International Symposium on Computer Architecture*, pages 297–306, 1986.

[80] Tim Wilkinson. KAFFEE: A free virtual machine to run Java code. http://www.kaffee.org.

[81] Ifor Williams and Mario Wolczko. An Object-Based Memory Architecture. In *Proc. Fourth International Workshop on Persistent Object Systems*, pages 114–130, 1991.

[82] A. Wolfe. First Java-specific chip takes wing . *Electronic Engineering Times*, 22 April 1997.
http://www.techweb.com/wire/news/1997/09/0922java.html.

[83] T. Yeh and Y. Patt. A Comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 257–266, 1995.

[84] Y.N. Patt, S.W. Melvin, W.-M, Hwu and M.C. Shebanow. Critical Issues Regarding HPS, A High Performance Microarchitecture. In *Proceedings of Micro-18*, pages 109–116, 1985.

[85] Y.N. Patt, S.W. Melvin, W.-M. Hwu, M.C. Shebanow, C. Chen and J. We. Run-Time Generation of HPS Microinstructions from a VAX Instruction Stream. In *Proceedings of Micro-19*, pages 109–116, 1986.

[86] Y.N. Patt, W.-M. Hwu and M.C. Shebanow. HPS, A New Microarchitecture: Rationale and Introduction. In *Proceedings of Micro-18*, pages 103–108, 1985.

Vita

Ramesh Radhakrishnan was born in Trivandrum, India on November 15, 1973, the son of Radhakrishnan Nair and Chandrika Nair. After completing his high school education at St. George Grammar School in Hyderabad, India, he entered The College of Engineering, Trivandrum (Kerala University) in 1991. He recieved the degree of Bachelor of Technology in Computer Science and Engineering from The College of Engineering, Trivandrum, in October, 1995. He joined the graduate program for Computer Science and Engineering at The University of South Florida, Tampa in January 1996. He recieved the degree of Master of Science in Computer Science and Engineering from The University of South Florida in December 1997. In August, 1997, he entered The Graduate School at The University of Texas.

Permanent Address: Sree Vidya, Kalyani Mandiram Compound
Vazhutacaud, Trivandrum, Kerala
India, 695010.

This dissertation was typeset with LaTeX[1] by the author.

[1] LaTeX is an extension of Donald Knuth's TeXprogram.