國立交通大學

資訊工程學系

碩 士 論 文

應用在 ARM/Thumb 雙指令集處理器的嵌入式混合模式爪哇虛擬機器之設計與實作

Design and Implementation of Embedded Mixed-Mode JVM for ARM/Thumb Dual Instruction Set Processor

研 究 生：黃健豪

指導教授：單智君 博士

中 華 民 國 九 十 三 年 六 月

應用在 ARM/Thumb 雙指令集處理器的
嵌入式混合模式爪哇虛擬機器之設計與實作

Design and Implementation of Embedded Mixed-Mode JVM for ARM/Thumb Dual Instruction Set Processor

研 究 生：黃健豪　　Student：Jiann-Haur Huang

指導教授：單智君 博士　　Advisor：Dr. Jean Jyh-Jiun Shann

國 立 交 通 大 學
資 訊 工 程 學 系
碩 士 論 文

A Thesis
Submitted to Department of
Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master
In
Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

應用在 ARM/Thumb 雙指令集處理器的嵌入式混合模式爪哇虛擬機器之設計與實作

學生：黃健豪　　　　　指導教授：單智君　博士

國立交通大學資訊工程學系碩士班

摘要

用在桌上型電腦環境的爪哇虛擬機器，由於需要快速的執行效能，通常會採用即時編譯器作為執行的引擎。而隨著手機和個人數位助理 (PDA) 等智慧型行動裝置愈來愈普及，其應用的需求也逐漸朝向高效能來發展。有鑑於此一趨勢，研究如何在這種嵌入式環境中提昇爪哇虛擬機器的效能，便成了一個有趣的議題。在本研究中，有別於一般採用全功能即時編譯器的方式，我們設計並且實作了一個輕量級的即時編譯器，其架構在以直譯器為基礎的嵌入式爪哇虛擬機器上，而整個虛擬機器是以混合執行的方式在運作。透過此種設計方式，可以將即時編譯器所必須額外付出的程式空間減到最小。

除了在即時編譯過程中運用多項加速技巧以外，我們的嵌入式爪哇虛擬機器也利用到了一項硬體架構所提供的特色─雙指令集。大多數的嵌入式處理器都有提供此功能，主要是為了在執行效能與程式空間之間達到一個平衡點。藉由設定不同的組態並作實驗評估，我們發現採用 ARM 直譯器並搭配標的為 Thumb 的即時編譯器，在同時考量效能和程式空間之下，可以達到比較好的效果。整體而言，我們的虛擬機器和單純 ARM 直譯器的虛擬機器作比較，效能是它的 2.08 倍，且只需額外付出 10.18% 的程式空間；而和單純 Thumb 直譯器的虛擬機器相比，效能是它的 3.21 倍，且只需額外付出 27.41% 的程式空間。

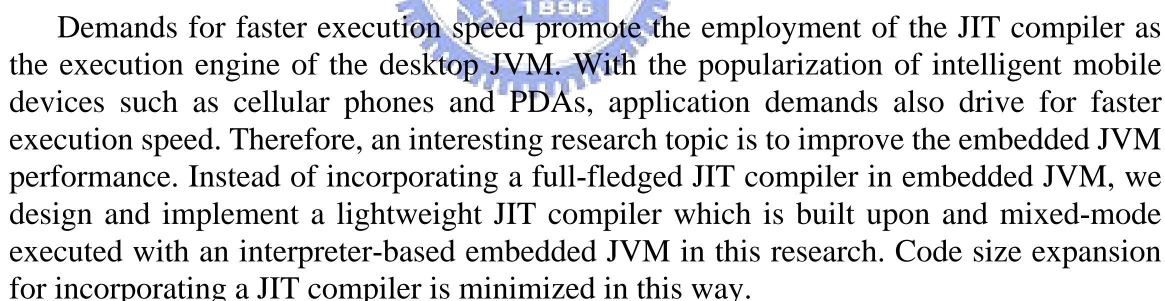
Design and Implementation of Embedded Mixed-Mode JVM for ARM/Thumb Dual Instruction Set Processor

Student: Jiann-Haur Huang Advisor: Dr. Jean, J.J. Shann

Department of Computer Science and Information Engineering

National Chiao-Tung University

Abstract



Demands for faster execution speed promote the employment of the JIT compiler as the execution engine of the desktop JVM. With the popularization of intelligent mobile devices such as cellular phones and PDAs, application demands also drive for faster execution speed. Therefore, an interesting research topic is to improve the embedded JVM performance. Instead of incorporating a full-fledged JIT compiler in embedded JVM, we design and implement a lightweight JIT compiler which is built upon and mixed-mode executed with an interpreter-based embedded JVM in this research. Code size expansion for incorporating a JIT compiler is minimized in this way.

In addition to employing several optimization techniques during JIT compilation, our embedded JVM also facilitate the "dual instruction set", an architectural feature that most embedded processors provide, in order to strike a balance between speed performance and code size. By setting up different configurations for evaluation, our experiments show that the ARM interpreter and Thumb JIT compiler is the most cost-effective configuration among the all. As a whole, our system demonstrates 2.08 speedup with only 10.18% code size increment over a pure ARM interpreter and 3.21 speedup with only 27.41% code size increment over a pure Thumb interpreter.

誌謝

首先必須向我的指導老師 單智君教授，獻上我最誠摯的謝意。在老師諄諄教誨、辛勤的指導之下，我得以完成此論文，並且順利通過畢業口試。同時感謝實驗室的另一位大家長 鍾崇斌教授，多次提出批評與指正，使論文得以更為嚴謹。再者，感謝校外口試委員 李政崑教授，在口試時提供許多寶貴的意見，使得這篇論文更加完整，而我本人也受益良多。

此外，我也很感謝宋宜叡同學，在與他合作 Mini-JIT 計劃的期間，透過相互的討論與腦力激盪，彼此在研究上都受益匪淺。接者，感謝 Java 組的喬偉豪學長，以及黃欽毓、黃俊諭、陳裕生、劉彥志等四位學弟，對於我的研究提出問題並給予建議。還有，感謝實驗室的全體學長姐、同學、以及學弟們，你們的陪伴使我的研究生活更加充實與豐富。

最後，感謝我的家人默默地給予我支持和鼓勵，讓我可以堅持追求自己的理想，在兩年的碩士生涯裡投入於課業以及論文研究之中！

謹向所有支持我、勉勵我的師長與親友，奉上最誠摯的祝福。謝謝你們！

黃健豪

2004.7.12

Contents

摘要 i

Abstract ii

誌謝 iii

Contents iv

List of Figures vi

List of Tables vii

Chapter 1 Introduction 1

1.1 Embedded Java Environment 1
1.2 Embedded Mixed-Mode Execution JVM 3
1.3 Dual Instruction Set For Code Size Reduction 4
1.4 Research Motivation and Objectives 5
1.5 Organization of This Thesis 5

Chapter 2 Background 6

2.1 Java Technology 6
2.1.1 JVM Benefits 6
2.1.2 JVM Internals 8
2.1.3 JVM Implementation Alternatives 10
2.2 JIT Compiler Optimizations 11
2.2.1 Common Optimization Techniques 11
2.2.2 Optimization Range 14
2.3 Related Researches 15
2.3.1 Dual Instruction Set 15
2.3.2 Embedded JVM 16

Chapter 3 System Design 18

3.1 System Overview 18
3.2 Speed Performance Analysis 21
3.2.1 Interpreter-Based JVM 22
3.2.2 Mixed-Mode Execution JVM 23
3.2.3 Speedup of Mixed-mode Execution Over Interpreter-Execution 24

3.3 KJITC Architecture 26
3.3.1 IR Generator. 28
3.3.2 Native Code Generator. 32
3.4 KJITC Optimizations. 33
3.4.1 Instruction Folding For Stack Operations 33
3.4.2 Rule-based Null Pointer Check Elimination 36
3.5 ARM/Thumb Instruction Set Selection 40

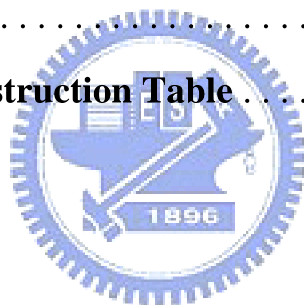
Chapter 4 Experiments. 42

4.1 Experiment Environment. 42
4.2 Benchmarks 42
4.3 Experiment Results 43
4.3.1 Effects of KJITC Optimizations 43
4.3.2 Effects of Dual Instruction Set Selection. 45

Chapter 5 Conclusion and Future Work 49

References. 51

Appendix Bytecode Instruction Table 54



List of Figures

Figure 1-1. Java 2 Platform 2
Figure 2-1. JVM Runtime Environment 8
Figure 2-2. Three Alternatives to Executing Java Programs 10
Figure 2-3. A Constant Folding Example 11
Figure 2-4. A Copy Propogation Example
(a) Before Copy Propogation (b) After Copy Propogation 12
Figure 2-5. An Example of CSE 12
Figure 2-6. An Example of Scalar Replacement and
Common Effective Address 13
Figure 3-1. System Components and Their Interactions 18
Figure 3-2. System Flowchart 19
Figure 3-3. An Illustration of KJITC 21
Figure 3-4. The Interpreter Dispatch Loop 22
Figure 3-5. Timing Diagram of the Interpreter-based JVM 23
Figure 3-6. Timing Diagram of the Mixed-mode Execution JVM 23
Figure 3-7. The Trend of Speedup 26
Figure 3-8. Two-pass Compiler Architecture 26
Figure 3-9. One IR Generator With Many Native Code Generators 27
Figure 3-10. The Frame Structure in Memory 29
Figure 3-11. Input and Output of the IR Generator 31
Figure 3-12. Stack Operations (a) Without Folding (b) With Folding 34
Figure 3-13. IR Generation
(a) Without Optimization (b) With Instruction Folding 34
Figure 3-14. Instruction Folding for Stack Operations During Code Generation 35
Figure 3-15. Flowchart of Null Pointer Check Elimination 38
Figure 4-1. Effects of Optimizations 44
Figure 4-2. Speed Performance of All Configurations 46
Figure 4-3. Compilation Cost of KJITC 46
Figure 4-4. Static Memory Usage of All Configurations 47
Figure 4-5. Dynamic Memory Usage of the Two JIT Compilers 47
Figure 4-6. Speed Increment and Code Size Increment of Four Mixed-mode
Configurations 48

List of Tables

Table 1-1. J2ME Configurations 2
Table 2-1. Comparison Among Some JIT Compilers 17
Table 3-1. An Example of Rule-based Null Pointer Check Elimination..............39
Table 3-2. Immediate Fields of Major Instruction Types....................................41
Table 4-1. Selected Tests of Embedded CaffeineMark 3.0..................................43
Table 4-2. Execution Cycles of Different Setups 44
Table 4-3. Execution Cycles of Six Configurations 45
Table 5-1. Comparison of KJITC with Other JIT Compilers...............................50

Chapter 1 Introduction

In this chapter, some introduction materials are presented to help readers understand the essential concepts behind and the terms in the title of our research. First, we give an overview of the current status of the Java technology in embedded environment. Second, we explain the meaning of mixed-mode, which actually combines interpretation and just-in-time (JIT) compilation, and the reason it suits for embedded JVM. Third, we discuss dual instruction set, an issue that is specifically relevant to embedded processors. After the introduction comes our research motivation and objectives. Finally, organization of this thesis is provided.

1.1 Embedded Java Environment

Developed by Sun in 1991, Java technology has evolved rapidly and becomes popular in all application fields, such as desktop PCs, powerful large-scale server, or even in small portable consumer devices. Recognizing the fact that different application fields possess different characteristics and demands, Sun in 1999 has grouped Java technologies into the Java 2 platform [1], which consists of three editions as in Figure 1-1, and each of which aims at a specific area:

- Java 2 Enterprise Edition (J2EE) - targeted at scalable, transactional, and database-centered enterprise applications with an emphasis on server-side development.
- Java 2 Standard Edition (J2SE) - targeted at conventional desktop applications.
- Java 2 Micro Edition (J2ME) - targeted at embedded and consumer devices, such as wireless handhelds, PDAs, TV set-top boxes, and other devices that lack the resources to support full J2SE implementation.

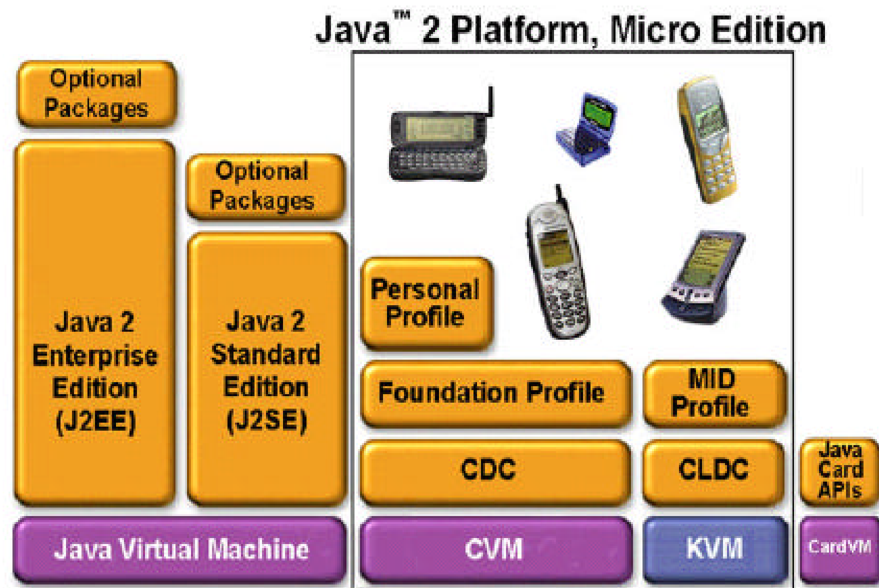


Figure 1-1. Java 2 Platform (extracted from Sun)

To address the diversity of large embedded world, which covers a wide range of devices, J2ME specifies two configurations: Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC). Each configuration targets at different types of devices and therefore provides different class libraries and APIs. Table 1-1 gives an overview of the differences of the two configurations.

Table 1-1. J2ME Configurations

Configurations Name	**Connected Device Configuration (CDC)**	**Connected Limited Device Configuration (CLDC)**
Target Devices	high-end PDAs, set-top boxes, screen phones, and etc.	cell phones, two-way pagers, low-end PDAs, and etc.
Typical Memory Requirement	2MB~16MB	128KB ~ 512KB
Target Processor Type	32-bit	16-bit, 32-bit
Reference Virtual Machine	CVM	KVM
Other Features	high bandwidth network connection, most often based on TCP/IP	limited, low bandwidth network connection

1.2 Embedded Mixed-Mode Execution JVM

Although the JVM can be easily realized by an interpreter, its slow performance is always a concern in performance-aware system. In order to overcome this problem, some compilation technologies must be applied. Ahead-of-time (AOT) compilers [2] allows off-line compilation, so no run-time compilation overhead is needed. Conventional JIT compilers translate bytecode into machine code on the fly, and incorporate more optimization techniques for better performance with the expense of VM code size increase and run-time compilation overhead. However, memory-constrained JVM can tolerate neither the static compiled code size expansion imposed by AOT compilers nor the code size/compilation overhead imposed by conventional JIT compilers.

The approach of mixed-mode execution in [3][4] relies an interpreter to execute interpreted code for some parts of the program, and also executes compiled code dynamically produced by a JIT compiler for the remaining parts. The line between a conventional JIT compiler and a JIT compiler that supports mixed-mode execution is, in actuality, indistinct. Nevertheless, the principles of mixed-mode execution can be clarified as follows.

- Performance-critical parts of the program are compiled by a JIT compiler, and then natively executed.
- Non-performance-critical parts of the program are interpreted by a interpreter.
- Close interactions between the JIT compiler and the interpreter is necessary.

As discussed in Section 1.1, embedded JVM (including its class libraries) has very limited memory budget, usually in the range of hundreds of kilobytes. For this reason, embedded JVM usually employs merely an interpreter as its execution engine. But with the increasing demands for speed performance, embedded JVM also seeks ways to improve its slow execution speed. The most effective way is to incorporate a JIT compiler, as most desktop/server JVMs do. Still, taken limited memory resources into consideration, a full-fledged JIT compiler does not suit for an embedded JVM. Therefore, a lightweight JIT compiler, which is highly-customized for an embedded JVM, is needed. To this end, a mixed-mode JVM seems to be promising in embedded environment. By tightly coupling

with an interpreter, a JIT compiler can reuse the interpreter-based JVM as its infrastructure, in order to keep itself compact. And overall the combination (an interpreter-based JVM and a JIT compiler) builds up an embedded mixed-mode JVM.

1.3 Dual Instruction Set For Code Size Reduction

Due to the requirements of low manufacturing cost, low power consumption, and small volume size, embedded systems usually have limited hardware resources, especially in memory size. 8-bit, 16-bit MCU processors have dominated the embedded system for a long time. However, with the increasing demands on more data applications in high-performance embedded system, 32-bit embedded processors have become mainstream these days.

Most 32-bit embedded processors are RISC-based, which suffer from the problem of poor code density and thus require more memory space. This is a severe limitation for cost-sensitive embedded systems. An innovative solution in architectural level is to employ “dual instruction set” [5]. One, the full instruction set, contains original 32-bit instruction set; the other, the compressed instruction set or the reduced bit-width instruction set, encodes most commonly used instructions in fewer bits (usually 16 bits).

According to previous researches, a program compiled in compressed instruction set will be much smaller than that in full instruction set. For example, the code size reduction of Thumb/ARM is 30% [6], while the case of MIPS16/MIPS32 is 30%~40% [7]. However, due to a limited set of instructions and access to a limited set of registers, a program will be compiled into more instructions in the compressed instruction set, which may result in overall performance degradation. Therefore, how to effectively facilitate dual instruction set to keep a balance between code size and performance, is both a practical industrial problem and a hot research topic.

1.4 Research Motivation and Objectives

Our observations are that while an embedded JVM manages to improve its execution speed, it still faces the problem of limited memory resources. Motivated by this fact, our objective is to design and implement an embedded JVM, which is small footprint compared to other existing embedded JVMs. We employ mixed-mode execution in our embedded JVM and further facilitate the "dual instruction set" feature that hardware architectural provides, aiming at striking a balance between speed performance and memory usage.

In addition, some practical decisions of our research are listed as follows.

- Our focus is on the design and implementation of a baseline JIT compiler for an embedded mixed-mode JVM, based on Sun's CLDC KVM 1.0.4 (interpreter-based). For ease of reference, the JIT compiler is hereafter termed KJITC, an abbreviation for "Kilobyte Just-In time Compiler".
- KJITC targets the ARM/Thumb dual instruction set processor.

1.5 Organization of This Thesis

The remaining parts of this thesis is organized as follows. Chapter 2 provides more detailed background knowledge on JVM internals and common JIT compiler optimizations. In Chapter 3, the design of KJITC is presented along with speed performance analysis and the design of ARM/Thumb instruction setction. In Chapter 4, experiemnt results are exhibited. In the end we make a brief summary in Chapter 5.

Chapter 2 Background

This chapter provides more background details on JVM internals and JIT compiler optimizations. Readers who are already farmiliar with the two topics can skim over them. Also some related researches on dual instruction set and embedded JVM are discussed in the last section.

2.1 Java Technology

Although generally used to refer to a computer language, Java is a rather a complete architecture in reality. It consists of four distinct but interrelated components [8].

- Java programming language
- Java class file format
- Java Application Programming Interface (Java API)
- Java Virtual Machine (JVM)

A Java program is written in Java programming language, and then compiled into Java class files by Java source compiler. Java class files can be executed on any environment that equips a JVM. Also, the Java program can access predefined libraries or system resources (such as I/O, for example) by calling methods in the classes that implement the Java API. During program execution, JVM loads and executes user-written class files as well as these system classes that Java API defines.

2.1.1 JVM Benefits

Java Virtual Machine is definitely the key component among the all. It is responsible for the well-known advantages that Java possesses over traditional native execution system. Those advantages include:

- Cross-Platform Portabilty

Each type of processor has its unique instruction set. For example, the instruction set of x86 is not compatible with that of MIPS. Moreover, each operating system (OS) has its own application interface or system calls to upper application programs. As a result, programs compiled to run on one platform (combination of processor and OS) cannot be executed on others without recompilation. Java overcomes this limitation by inserting JVM between the application programs and the real environment. If JVM has been ported to the environment, Java programs can be first compiled to Java bytecode in the form of class files and then be executed over the JVM without any porting efforts. This encourages software reuse and alleviates great pains from programmers.

- Security of the Execution Environment

One of Java's original intention is its integration into the network environment. In this environment, class files can be automatically downloaded from network and be locally executed. They might be malicious and might do dangerous operations to the local execution system. To deal with this important issue, Java build up its own security model - the sandbox [9][10]. As a brief explanation, Java verifies every class file from untrusted resources. The verification process mainly involves two steps in JVM. First, class file verification checks the layout and the contents of the class file. Second, bytecode verification checks if the bytecode within a method adheres to predefined rules. For example, one basic rule is that all goto and branch instructions refer to valid bytecode addresses.

- Small Size of the Compiled Code

Due to the rich semantics and the stack-based operations, Java bytecode, the instruction set of JVM, is more compact space-wise than a statically compiled program. In other words, Java has high code density. According to [11], the dynamic average instruction size is 1.8 bytes. Compared with typical RISC instruction requiring 4 bytes, this result is satisfactory. For a speed-limited network environment or a memory constrained embedded

environment, small code size is undoubtedly favorable.

2.1.2 JVM Internals

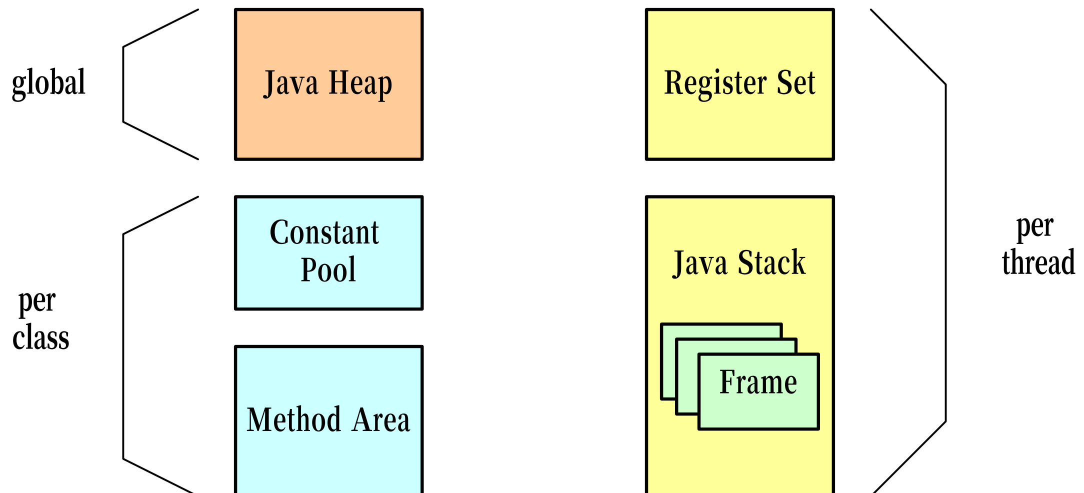


Figure 2-1. JVM Runtime Environment

To realize the JVM, an implementation must provide the functionality of a real processor and also adhere to the JVM specification [12]. The specification defines a homogeneous run-time environment, as Figure 2-1 illustrates, by providing a detailed description of the following items:

- Instruction Set (Java Bytecode)
- Register Set
- Java Stack
- Execution Environment
- Constant Pool
- Method Area
- Java Heap
- Object Management and Garbage Collection

Since the JVM is a stack-based architecture, the registers of its register set are not used for storing operands or passing arguments as in most register-based machine. They only hold the state of the JVM and are updated after every bytecode instruction is executed.

The operands of a bytecode instruction must be pushed onto the Java stack before the instruction is executed. An executing instruction consumes its operands from the stack and then places results on the stack when it completes.

The execution environment is maintained within the Java stack as a data set and is used to deal with dynamic linkage, method invocation/return and exception handling. It handles dynamic linkage by maintaining symbolic references to methods and variables for the current method and current class. A symbol table is used to translate these references to actual calls.

The JVM maintains a special table for each class, known as a constant pool. The constant pool contains string literals, class names, field names and other constant data objects that are referred to by the class structure or by the executing program. These constants do not change, and 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 begin in the class description.

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

The Java heap is the dynamic memory of JVM, and it usually contains a collection of objects. When an object is created with the "new" bytecode instruction, an reference to that object is returned. This reference can be used subsequently, or stored in the current frame. An object persists in Java heap 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 collector will reclaim its resources.

2.1.3 JVM Implementation Alternatives

The JVM is not restricted to software interpreter implementation. In fact, there are three common approaches, as depicted in Figure 2-2, to implement the JVM.

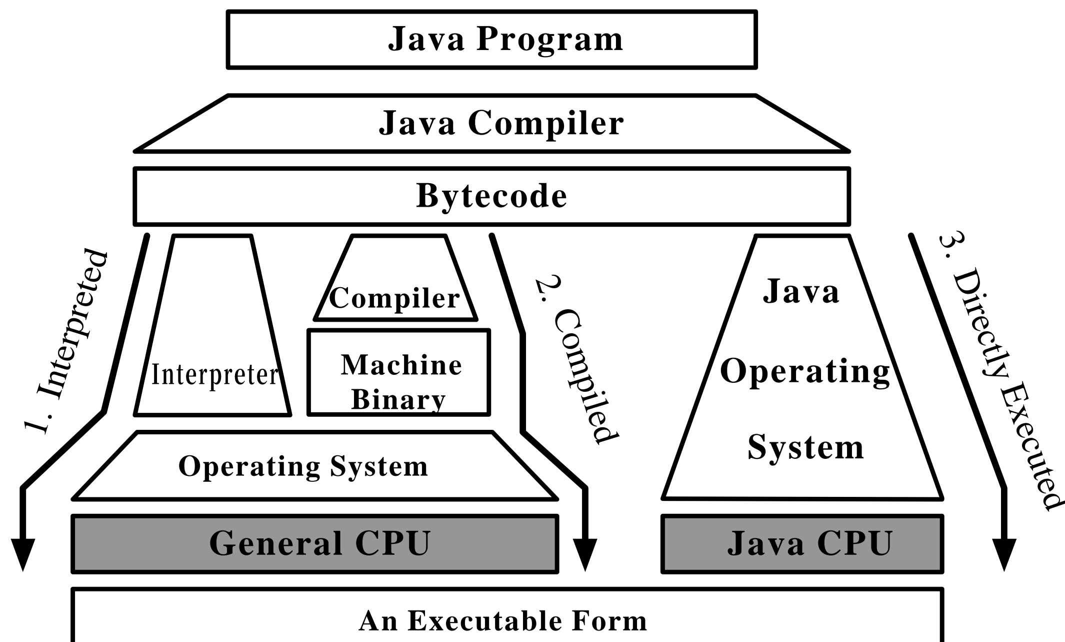


Figure 2-2. Three Alternatives to Executing Java Programs (extracted and modified from [13])

Interpreting the bytecode, the standard way to implement the JVM, has the advantage of fast JVM porting but makes the execution of Java programs relatively slow. One solution to improve speed performance is to replace an interpreter with a bytecode compiler. The bytecode compiler is responsible for translating bytecode into native machine code. While ahead-of-time (AOT) compilers performs offline compilation statically as conventional compilers, just-in-time (JIT) compilers performs on-the-fly compilation dynamically. Both of them have pros and cons, but JIT compilers seem to be more appealing to most researchers. Another solution is to implement the JVM directly on silicon. For example, picoJava is a Java processor that supports bytecode execution completely.

As discussed in Section 1.2, an interpreter can still coexist and cooperate with a JIT compiler in the JVM. Recently, a mixed software/hardware approach also comes to exist. ARM has introduced its own Java instruction extension - Jazelle [14]. A subset of bytecode

instructions can be directly executed when the ARM processor is operated in Java mode, and the remaining bytecode instructions are still handled in software (interpreted or compiled).

2.2 JIT Compiler Optimizations

Since JIT compilers perform compilation at run time, the restriction of compilation time is more severe than that in traditional static compilers. As a result, only cost-effective optimization techniques can be suitably applied during JIT compilation. Due to the characteristics of Java, optimization techniques might cause different impact when applied in Java JIT compilers than in traditional static C compilers. In this section, we are to discuss some common optimization techniques used in Java JIT compilers [15][16][17], and then to discuss different ranges of optimization.

2.2.1 Common Optimization Techniques

Constant Folding

The concept behind constant folding is to evaluate constant expression, whose operands are known to be constant, at compile time. After this simple transformation, the constant expression is replaced by its value. Therefore it saves the run-time computation of the expression. A simple example is demonstrated in Figure 2-3.

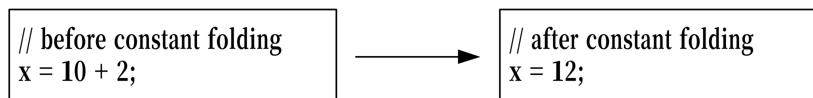


Figure 2-3. A Constant Folding Example

Copy Propagation

Copy propagation is a transformation that replace variable occurrences with its copy value which is defined in earlier copy assignments. For example, the copy assignment is represented in the form $x = y$, for some variables x and y. Then later uses of x, as long as intervening instructions have not changed the value of either x or y, can be replaced with y.

Figure 2-4 is an example in the flowgraph form. The copy assignment is $b = a$, and succeeding occurrences of b of the underlined expressions are replaced with a.

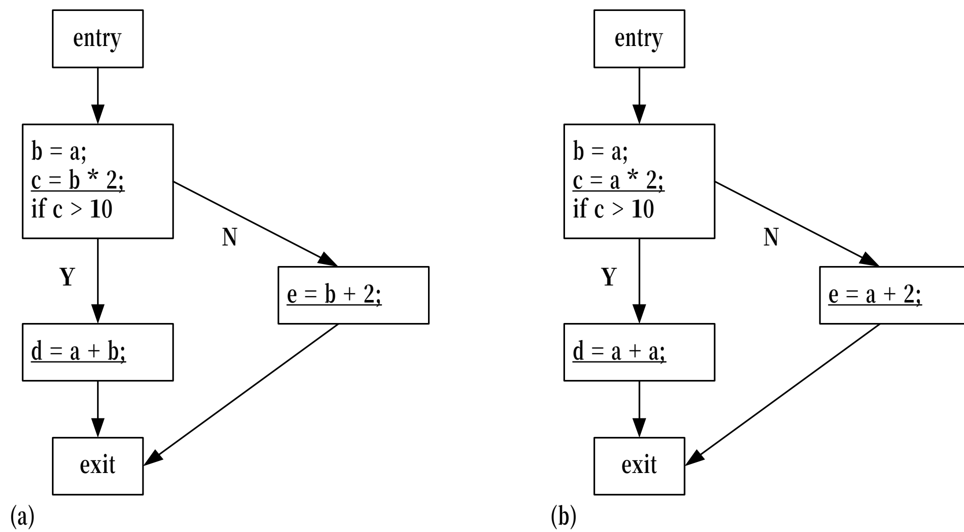


Figure 2-4. A Copy Propogation Example
(a) Before Copy Propogation (b) After Copy Propogation

1896

Common Sub-expression Elimination (CSE)

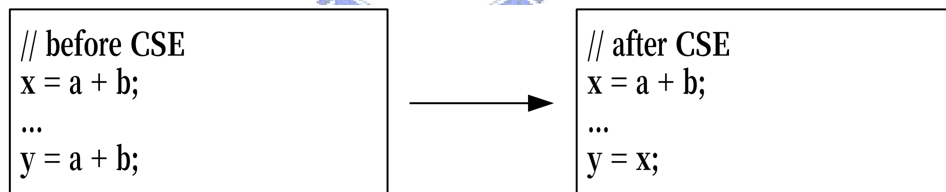


Figure 2-5. An Example of CSE

The purpose of CSE is to reduce repetitive computations by substituting available results for the expressions that do the same computation. Figure 2-5 gives a simple example. Also two common derivatives of CSE are:

- Scalar Replacement

Array element accesses in a loop are replaced by temporary variables, when the array objects and the array indexes remain unchanged. See the example in Figure 2-6.

• Common Effective Address Generation

Successive array element accesses in a loop can be optimized by introducing a temporary pointing to the first element. Therefore other elements can be accessed by using the temporary as the base address and corresponding array indexes as offsets. See the example in Figure 2-6.

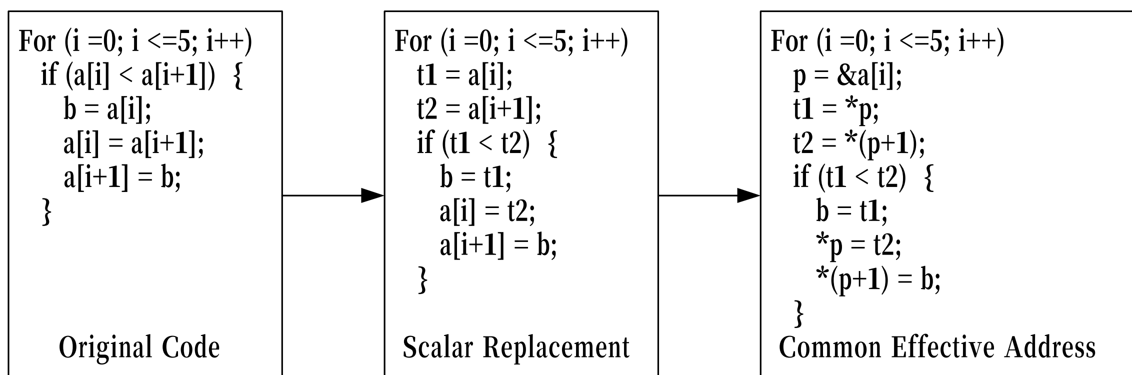


Figure 2-6. An Example of Scalar Replacement and Common Effective Address

Exception Check Elimination

Java bytecode instructions contain semantics that may induce exceptions. In an interpreter, such bytecode instructions are checked during interpretation to see if exceptions arise. If they are, appropriate exception handlers are invoked. For a JIT compiler, to compile these bytecode instructions also produce compiled code that performs exception check. However, some of these checks are redundant and can be eliminated via careful analysis. In short, exception check elimination helps to save unnecessary operations and also reduce code size. Null pointer check elimination and array bound check elimination are the most common techniques used in Java JIT compilers.

Method Inlining

The idea of method inlining is to inline method calls by expanding method bodies. This optimization can reduce method invocation overhead in sacrifice of code size expansion and also can provide more optimization opportunities. In object-oriented languages like Java, tiny methods such as class constructors and methods that accesses private variables are frequently executed. These methods spend more time on method invocation than

method body execution. Hence method inlining is useful under these circumstances. Moreover, concerning the heavy overhead of devirtualization, virtual method calls may be inlined as well. Certainly, it involves further analysis.

Strength Reduction and Machine Idioms

Strength reduction is to replace an operation with a semantically equivalent one, though weaker but faster. A common case is using the shift operator to multiply and divide integers by a power of 2. For example, $x >> 2$ can be used in place of $x / 4$, and $x << 1$ replaces $x * 2$. In a similar way, machine idioms refer to instructions or instruction sequences for a specific ISA that executes more efficiently than a similar sequence of instructions targeted for a more general architecture. A good example is that some architectures provide multiply-and-add instructions for faster execution.

2.2.2 Optimization Range

Conventionally, an optimization applied to a program is generally called "local" if it is performed by looking only at the statements in a basic block; otherwise, it is called "global" [18]. To be more specific, "local" means optimization is applied within a basic block while "global" within a function. Some optimization techniques can be applied at both local and global levels. Global optimization invests more compilation time in advanced analysis, and therefore leads to better compiled code quality.

Local optimization might expand its optimization range from a basic block to an extended basic block [19]. As a contrast to single-entry-single-exit basic blocks, extended basic blocks are also single-entry but possibly multiple-exit, and therefore have more opportunities for optimization. Researches on high performance architectures focus on loop optimization in a program. In fact, high-level loop structures may be recovered by identifying strongly connected components (SCCs) or regions in a low-level control flow graph. Furthermore, interprocedural optimization is more aggresive for its range expands across functions, and thus is considered to be pretty costly. In short, as the optimization range is enlarged from local to loop and global, or even interprocedural, the cost of analysis definitely increases. For more detailed information, please also refer to [19].

2.3 Related Researches

This section briefly introduces the essentials about dual instruction set and its current research status. Next, advancements in optimization for embedded JVM is discussed as well, including one recent research work on embedded JIT compilation.

2.3.1 Dual Instruction Set

A number of 32-bit RISC processors for embedded systems may incorporate a reduced bit-width instruction set as an architectural extension, and therefore support dual instruction set. ARM provides its 16-bit instruction set extension called Thumb since its ARM7 processors in 1995. With a decompression engine, Thumb instructions are converted to its ARM equivalents during decode pipeline stage. Switching between the two instruction sets is achieved through the use of explicit mode change (ARM mode and Thumb mode) insructions. Thumb instructions are only able to access 8 general purpose registers (out of 16) without any restrictions, and can only encode small immediate values. Also addressing modes and instruction types are restricted in Thumb instruction set. Experiment results exhibit with 32-bit memory Thumb trades off 30% - 40% speed performance for 30% code size reduction.

MIPS follows ARM by offering its MIPS16 instruction set in 1997. As a contrast to Thumb, MIPS16 contains an extend opcode which extends the values of immediate operands that are not representable due to bit width constraints. Rather than switch with explicit mode change instructions, code alignment dictates the mode of execution. To be more specific, a function that is not word-aligned is assumed to be composed of MIPS16 instructions. Experiment results show the code size reduction is up to 40% using MIPS16. Other processors that support dual instruction set include the ST100 Core [20] from ST Microelectronics and the Tangent-A5 [21] from ARC.

Two recent research papers [22][23] about dual instruction set are on evaluation of mixed instruction set code in different granularities such as function levels and basic block levels. Their proposed heuristics for instruction set selection are static, profile guided and may be based on cost models. However, no apparent results can be inferred from the

researches about how to perform instruction set selection for specialized environments such as a mixed-mode JVM. This also serves as a reason that motivates us to conduct this research work.

2.3.2 Embedded JVM

Due to tight memory constraints, embedded JVM usually seeks its way for performance improvement by adopting low-cost optimizations in terms of code size. These low-cost optimizations manage to improve overall performance by reducing overheads in exception handling, garbage collection, object access and bytecode dispatch. Among of them, optimization for bytecode dispatch is most effective since dispatch time occupies a great portion of total execution time. Researches in [24][25] discuss different threading mechanisms to improve dispatch efficiency for JVM. Moreover a bytecode instruction sequence can be grouped together or formed into a new bytecode, and therefore only one dispatch is necessary as described in [26]. Since the sequence is executed as a whole, there are opportunities that it can be executed more efficiently by optimizing native code. Related works of this type include [27][28].

Although aforementioned optimizations can be employed in embedded JVM without much code size expansion, their performance improvement is potentially and relatively low compared with JIT compilation. As a result, for embedded JVM that demands high performance, JIT compilation is indispensable. A recent work [29] demonstrates a JIT compiler designed for employment in embedded JVM. Table 2-1, which is extracted and modified from the same work, lists some important features of this embedded JIT compiler compared with other JIT compilers. Apparently the embedded JIT compiler consumes much code size such that highly-memory-limited embedded systems can not afford. Therefore, there is still research space for more lightweight JIT compilation that can be applicable to a wider range of embedded systems with JVM.

Table 2-1. Comparison Among Some JIT Compilers

JIT	Sun - Server	Sun - Client	SNU Latte	Stanford MicroJIT
Source	C++	C++	C	C
IR Format	SSA dataflow	Simple	Dataflow	Dataflow
Major Compiler Passes	Interative	4	7	4
Register Allocation	Graph coloring	1-pass dynamic	2-pass dynamic	1-pass dynamic
Major Optimizations	1. loop invariant code motion 2. global value numbering 3. constant propagation 4. inlining & specialization 5. instruction scheduling	1. block merging/elimination 2. simple constant propagation 3. inlining & specialization	1. EBB value numbering 2. EBB constant propagation 3. loop invariant code motion 4. inlining & specialization	1. CSE 2. copy propagation 3. constant propagaron 4. loop invariant code motion 5. inlining & specialization 6. instruction scheduling
Compiler Size	1.5MB (Sparc)	700KB (Sparc)	325KB (Sparc)	200KB (Sparc)
Compilation Cost (Per Bytecode)	~100,000 Cycles	~8,300 Cycles	~20,000 Cycles	~5,000 Cycles

Chapter 3 System Design

In this chapter, we present the overall system design of our embedded mixed-mode JVM. Section 3.1 provides an overview of our system, which consists of four components, and then discusses their relative interactions. Section 3.2, a quantitative analysis on speed performance of our system, compared with that of a pure interpreter-based JVM, is proposed. The analysis helps us make our further design decisions in KJITC. Next, we detail the internal design of KJITC and its optimizations. Finally, we demonstrate design issues on ARM/Thumb instruction set selection.

3.1 System Overview

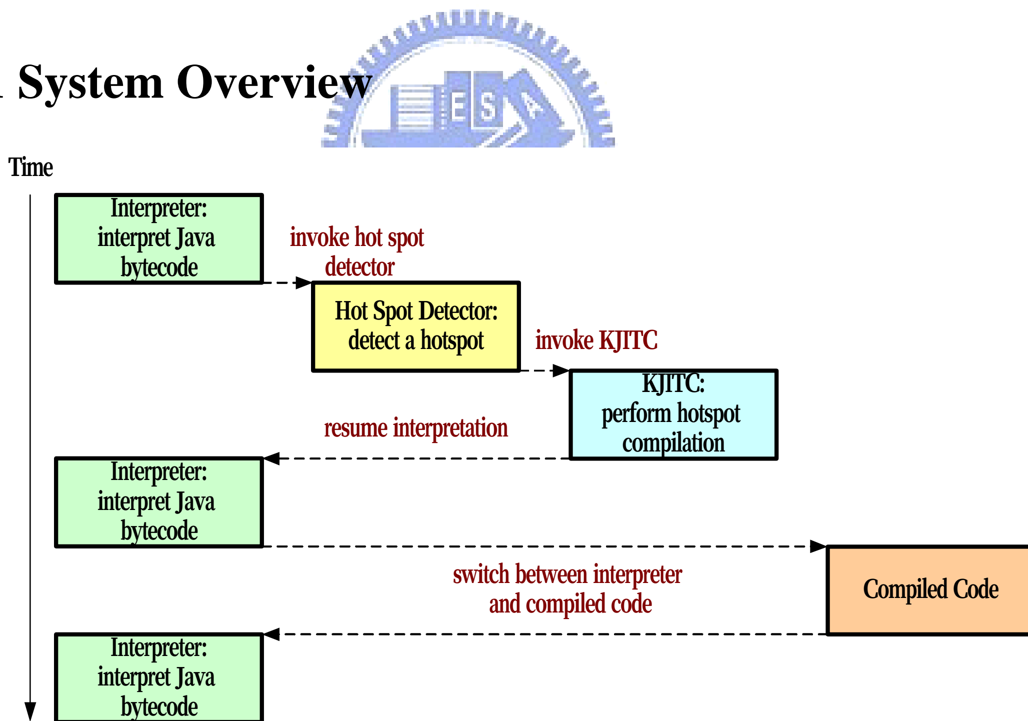


Figure 3-1. System Components and Their Interactions

In our mixed-mode embedded JVM, there are four main components. Their interactions can be simply illustrated in Figure 3-1. A more precise flowchart which describes the working flow during method interpretation is also provided in Figure 3-2 for completeness.

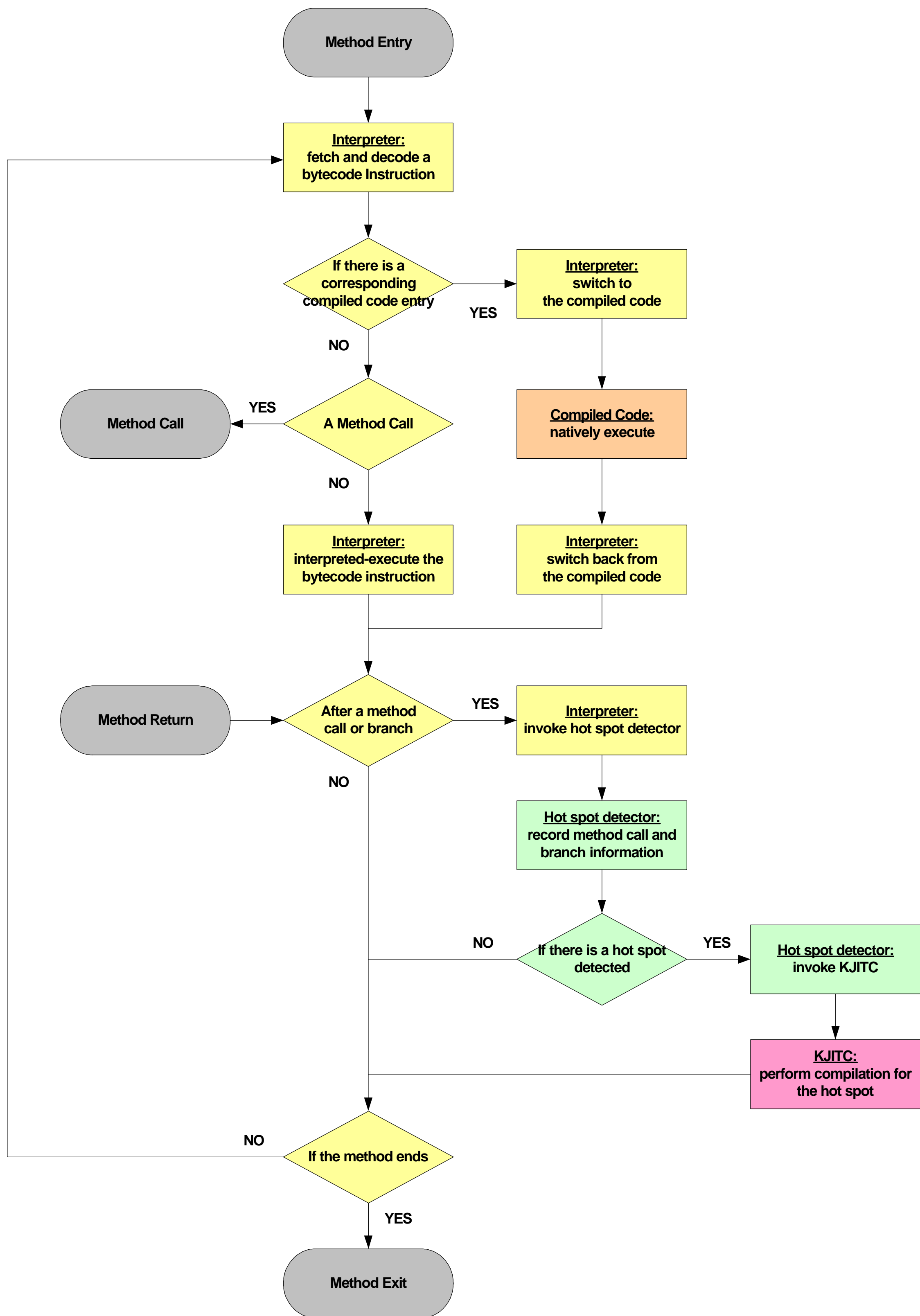


Figure 3-2. System Flowchart

Now we respectively discuss each component as follows.

- Interpreter-based JVM (KVM)

The interpreter-based JVM provides a JVM infrastructure that performs exception handling, garbage collection, synchronization and etc. It comes with a simple interpreter as its execution engine. For mixed-mode execution, the interpreter must be responsible for invoking the hot spot detector and switching to/form compiled code in addition to interpretation of those bytecode that have not been compiled or will not be compiled.

- Hot Spot Detector

Due to the tight memory constraints, only valuable parts of the input program are selected for JIT compilation. By the 80/20 rule, over eighty percent of execution time is spent in less than twenty percent of source code in a program. Apparently, the responsibility of the hot spot detector is to discover these performance-critical twenty percent of source code and then invoke JIT compiler for hot spot compilation.

As mentioned in Section 2.1.2, the method area is viewed as the run-time compiled code area. Hence we also select the method as the basic unit of hot spot detection. A method is considered to be a hot spot, when it meets either one of the following two requirements. First, it is called by other methods frequently. Second, it contains at least one loop that has many iterations. In our implementation, threshold values must be set statically as the criteria for the two requirements. Currently the values are both chosen to be 40, which are based on our evaluation results.

- JIT Compiler (KJITC)

The JIT compiler is further divided into the IR (Intermediate Representation) generator and the native code generator. The IR generator is mainly responsible for translating Java bytecode into semantically equivalent three-address IR. And then the code generator translates IR into targeted native code for later execution. A simple illustration is given in Figure 3-3.

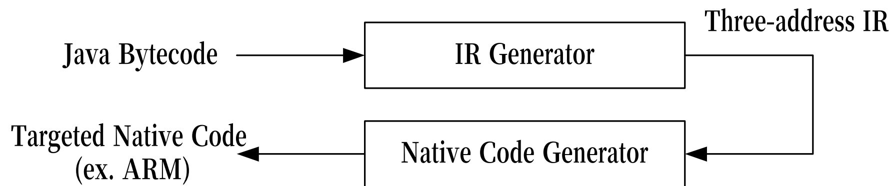


Figure 3-3. An Illustration of KJITC

- Compiled Code Buffer

The compiled code buffer holds all compiled native code. During native execution, the machine program counter (PC) points to native code that resides in the buffer. In our current implementation, the compiled code buffer is allocated statically, and its size is also pre-determined.

In addition to the four components, the switching mechanism between the interpreter and the compiled native code also deserves discussions. Similar to a function call, the switch from the interpreter to the compiled native code involves spilling registers into memory and then transfering execution by a branch. The case of the switch from the compiled native code to the interpreter involves more operations. It has to restore registers from memory, to transfer execution by a branch, and to update Java PC (program counter) and Java SP (stack pointer).

3.2 Speed Performance Analysis

Before proceeding to the focus of our research - the KJITC, we present basic quantitaive analysis of system performance in this section. First we begin with an interpreter-based system, and then compare it with our system, which exhibits mixed-mode execution.

3.2.1 Interpreter-Based JVM

```
NEXT:
switch (*bytecode_pc) {
  case ByteCode_1:
          …
          goto NEXT;
  case ByteCode_2:
          …
          goto NEXT;
  case ByteCode_3:
          …
          goto NEXT;
  case …
  ...
}
```

Figure 3-4. The Interpreter Dispatch Loop

Figure 3-4 shows a simplified dispatch loop - the main structure of an interpreter - in C language source form. An interprter may be viewed as a software processor that sequentally performs three tasks - fetching, decoding, and execution. For ease of reference and explanation, we delibrately break the total execution time of an interpreter-based JVM into the following three parts. Figure 3-5 is a timing diagram of an interpreter-based JVM which performs bytecode interpretation in a repetitive manner.

- Dispatch (fetching + decoding) time ... $\mathbf{T_{disp}}$
  => Dispatch time of a single bytecode instruction ... $\mathbf{t_{disp}}$
- Interpreter execution time ... $\mathbf{T_{int_exec}}$
  => Average interpreter execution time of a single bytecode instruction ... $\mathbf{t_{int_exec}}$
- Miscellaneous time ... $\mathbf{T_{misc}}$
  (garbage collection, synchronization, and etc.)

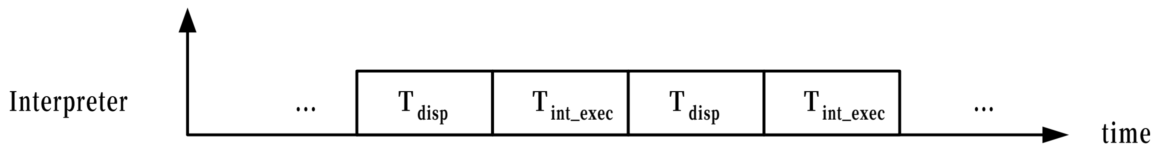


Figure 3-5. Timing Diagram of the Interpreter-based JVM

3.2.2 Mixed-Mode Execution JVM

Similarly, the breakdown of the total execution time in our mixed-mode execution JVM can be listed as the following six parts. Figure 3-6 is a typical timing diagram which comprises the leading five parts while omitting miscellaneous time for clarity.

- Dispatch (fetching + decoding) time ... $\mathbf{T'_{disp}}$
- Interpreter execution time ... $\mathbf{T'_{int_exec}}$
- JIT compilation time ... $\mathbf{T_{comp}}$
- Interpreter-native code switch time ... $\mathbf{T_{switch}}$ ($\mathbf{T_{switch_from}} + \mathbf{T_{switch_to}}$)
  => One switch time ... $\mathbf{t_{switch}}$ ($\mathbf{t_{switch_from}} + \mathbf{t_{switch_to}}$)
- Native code execution time ... $\mathbf{T_{native_exec}}$
  => Average native code execution time of a single bytecode instruction ... $\mathbf{t_{native_exec}}$
- Miscellaneous time ... $\mathbf{T'_{misc}}$

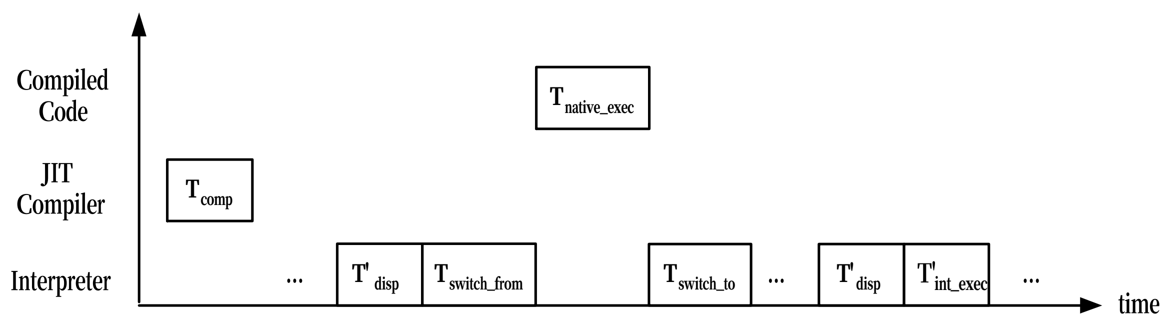


Figure 3-6. Timing Diagram of the Mixed-mode Execution JVM

3.2.3 Speedup of Mixed-mode Execution Over Interpreter-Execution

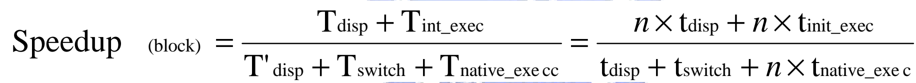
To compare relative performance of the mixed-mode execution JVM and the interpreter-based JVM, following speedup definition is provided.

$$\text{Speedup of A over B} = \frac{\text{Execution Time of B}}{\text{Execution Time of A}}$$

Then the speedup of the mixed-mode execution JVM over the interpreter-based JVM can be expressed as follows.

$$\text{Speedup}_{\text{(overall)}} = \frac{T_{disp} + T_{int_exec} + T_{misc}}{T'_{comp} + T'_{disp} + T_{int_exec} + T_{switch} + T_{native_execc} + T'_{misc}}$$

For a compiled sequence of *n* bytecode instructions, speedup can be approximated by the following equation.



$$\text{Speedup}_{\text{(block)}} = \frac{T_{disp} + T_{int_exec}}{T'_{disp} + T_{switch} + T_{native_exe\,cc}} = \frac{n \times t_{disp} + n \times t_{init_exec}}{t_{disp} + t_{switch} + n \times t_{native_exe\,c}}$$

For further analysis, some values in the above equation can be obtained in our implementation. Then the equation becomes:

$$\text{Speedup}_{\text{(block)}} = \frac{(21n) + T_{int_exec}}{(21) + (53) + T_{native_exe\,c}}$$

The meaning of the equation is that:

- It takes 21 cycles for every bytecode interpretation and some varied cycles for interpreted execution.
- It takes 21 cycles for identifying a sequence of compiled code, and the overall switching time is 53 cycles, plus some varied cycles for native execution.

With the equation, we make some discussions as follows.

- Since most bytecode instructions only involve simple operations, such as *IADD*, *ILOAD*, and *ISTORE*, the average interpreter execution time of a bytecode instruction is fewer than 21 cycles. As a reference, it is statically and roughly estimated to be 9.7 cycles in our implementation. According to Amdahl's law, the performance bottleneck is the dispatch time, $\mathbf{T_{disp}}$. Therefore, the first priority is to reduce the dispatch overhead by enlarging the value of *n*.
- The time of interpreted execution is definitely larger than that of native execution, for the JIT compiler can perform optimizations while the interpreter cannot. The value of $\mathbf{T_{int_exec}}$ over $\mathbf{T_{native_exec}}$ may be roughly referred as the code quality of the compiled code. In theory, when $n = 1$, the code quality shall be equal to 1, since there is no room for optimization. Conversely, when n grows larger, the code quality may grow larger as well. Therefore, the second priority is to improve the code quality, either by enlarging the value of *n* or by employing more optimizations.

As a motivating example, we consider the bytecode sequence of "*ILOAD_1, ILOAD_2, IADD, ISTORE_1*". The values of $\mathbf{T_{int_exec}}$ and $\mathbf{T_{native_exec}}$ can be simply computed in our implementation, assuming that instruction folding for stack operation in Section 3.4.1 is applied during compilation. Now the equation can be evaluated.

$$\text{Speedup}_{\text{(block)}} = \frac{(21 \times 4) + (33)}{(21) + (53) + (9)} = 1.24$$

From the above equation, we can compute the average values - $\mathbf{t_{int_exec}}$ and $\mathbf{t_{native_exec}}$, and then re-build up the speedup equation.

$$\text{Speedup}_{\text{(block)}} = \frac{(21n) + (6n)}{(21) + (53) + (2.25n)} = \frac{(27n)}{(74) + (2.25n)}$$

When the value of *n* equals to 20, the speedup is about 4.54. When the value of *n* equals to 50, the speedup is about 7.24. Figure 3-7 is a plot that exhibits the trend of speedup when *n* increases.

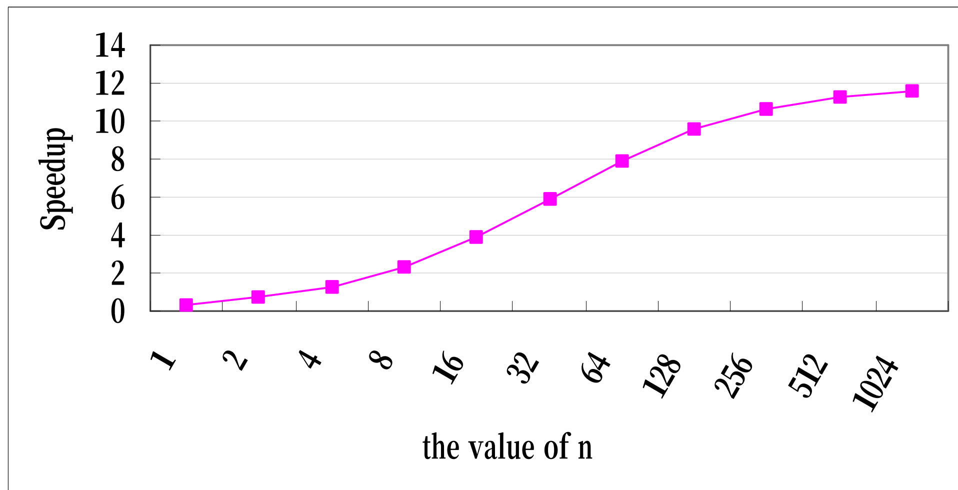


Figure 3-7. The Trend of Speedup

As the plot shown, the speedup will coverge as the value of *n* increases. Although this ideal speedup trend may differ from that in real cases, it still provides some useful guidelines when designing our baseline KJITC.

3.3 KJITC Architecture

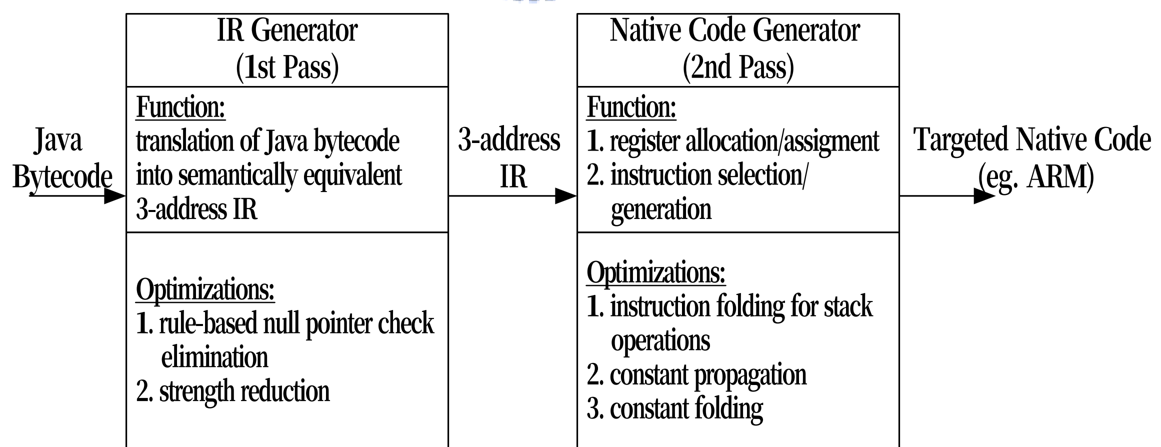


Figure 3-8. Two-pass Compiler Architecture

In this section, we detail the design of the IR generator and the native code generator in the KJITC. In addition, in order to reduce compilation cost and to keep the KJITC small-

footprint, several design decisions are made based on the analysis in Section 3.2. These decisions are:

- Two-pass Compiler Architecture

We confine our compiler to two passes. The first pass is for IR generation, and the second pass is for native code generation. Figure 3-8 gives a more detailed overview of functions and optimizations of the two passes. This decision is based on the fact that fewer passes take less compilation time and that two passes seem to be reasonable for portability. The IR generator is responsible for translating Java bytecode into machine-independent three-address IR, and therefore is portable across platforms. Clearly the KJITC needs only one IR generator while possessing more than one native code generator for different targeted architectures as depicted in Figure 3-9.

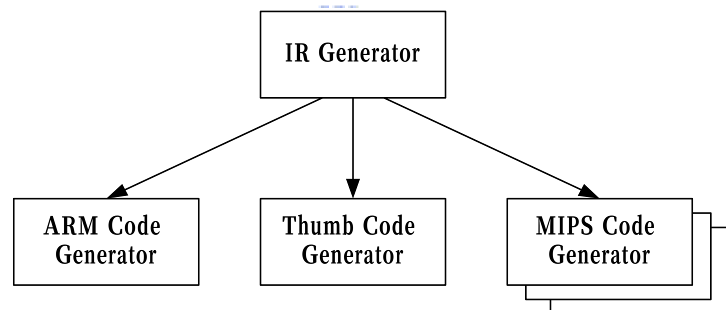


Figure 3-9. One IR Generator With Many Native Code Generators

- Only Local Optimization Within an Extended Basic Block

No global optimization is performed due to the potential high compilation cost of control and data flow analysis. However, we extend the maximum optimization range to an extended basic block rather than a basic block.

- Support for More Bytecode

If the KJITC can compile more types of bytecode, compilation may be possibly applied to a longer sequence of input bytecode, which in turn results in better performance as we have discussed in Section 3.2.3.

- No Support for Complex Bytecode

Complex bytecode refers to those bytecode instructions that involve complicated operations that suit for interpreter handling. These complicated operations include devirtualization, synchronization, object construction/destruction, and etc. As a result, these complex bytecode instructions are considered to be non-compile-able in the KJITC.

3.3.1 IR Generator

IR Format

The IR format is designed with the following two properties.

- Three Address Quadruple: (*Opcode*, *Arg1*, *Arg2*, *Arg3*)

Opcode refers to the instruction operation. *Arg1* generally refers to the destination of the operation. *Arg2* generally refers to the first source of the operation. *Arg3* generally refers to the second source of the operation.

- Local-Variable-Based Memory Addressing

Arg1, *Arg2*, and *Arg3* are used for storing constants or memory addresses. The memory addresses are local-variable-based. That is, the actual values stored are the offsets relative to the base address of the local variable array. In the KVM, the operand stack and the local variable array of a frame both reside in a linearly addressable range of memory, and thier relative addresses are also fixed (see Figure 3-10). During the execution of a program, frames are dynamically created and discarded, hence their memory addresses can only be determined at run-time. As a result, elements of the local variable array and the operand stack are addressed by using the starting address of local variable array as the implicit base address plus corresponding word-offsets encoded in instructions.

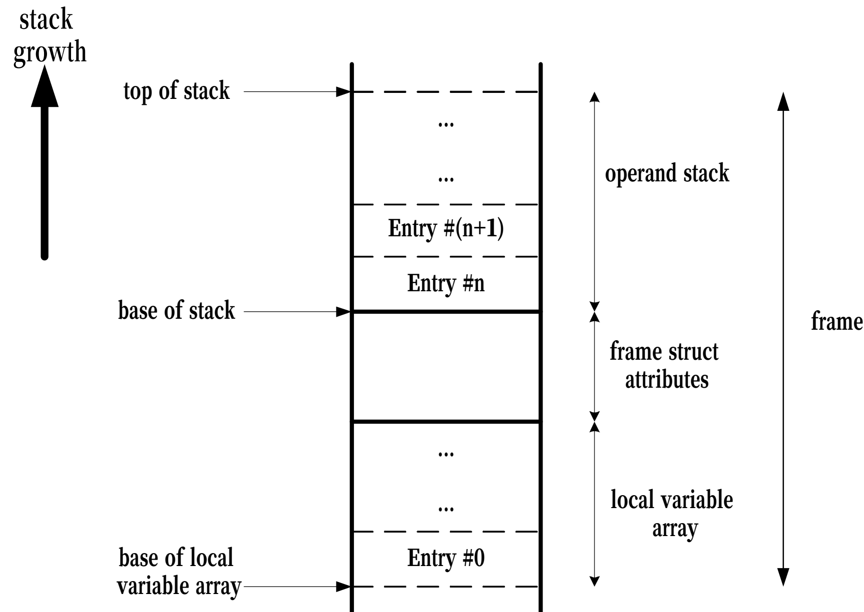


Figure 3-10. The Frame Structure in Memory

Bytecode to IR Translation

After the design of IR format is decided, bytecode can be easily translated into semantically-equivalent IR. Much of the work involves translation from implicit stack addresses into explicit local-variable-based addresses. Following are some examples for demonstration.

1. DUP

   • Bytecode Number: 89

   • Function: To duplicate the top element of the operand stack

   • Translated IR: (*MOV*, *&TOS[0]-&LV[0]*, ---, *&TOS[-1]-&LV[0]*)

   • Brief Description: The IR operation is *MOV*. The destination of the operation is the empty element of the operand stack. Since the top-of-stack pointer always points to the empty element of the operand stack, the destination can be addressed by *&TOS[0]-&LV[0]*. The first source of the operation is unused, and the second is the top element of the operand stack which is addressed by *&TOS[-1]-&LV[0]*.

2. ILOAD_1

• Bytecode Number: 27

• Function To push the second local variable onto the operand stack

• Translated IR: (*MOV*, *&TOS[0]-&LV[0]*, ---, *&LV[1]-&LV[0]*)

• Brief Description: The IR operation is *MOV*. The destination of the operation is the empty element which is addressed by *&TOS[0]-&LV[0]*. The first source of the operation is unused. The second source of the operation is the second local variable which is addressed by *&LV[1]-&LV[0]*.

3. IADD

• Bytecode Number: 96

• Function: to pop and add the top two elements from the operand stack, and then push the result back

• Translated IR: (*ADD*, *&TOS[-2]-&LV[0]*, *&TOS[-2]-&LV[0]*, *&TOS[-1]-&LV[0])*

• Brief Description: The IR operation is *ADD*. The destination and first source of the operation is the second top element of the operand stack which is addressed by *&TOS[-2]-&LV[0]*. The second source of the operation is the first top element of the operand stack which is addressed by *&TOS[-1]-&LV[0]*.

It is worth noting that a semantically-rich bytecode instruction may be decomposed into several simple IR instructions. For example, bytecode instructions for array access involve implicit exception checks, and therefore their decomposed IR instructions contain explicit exception checks.

IR Generation Workflow

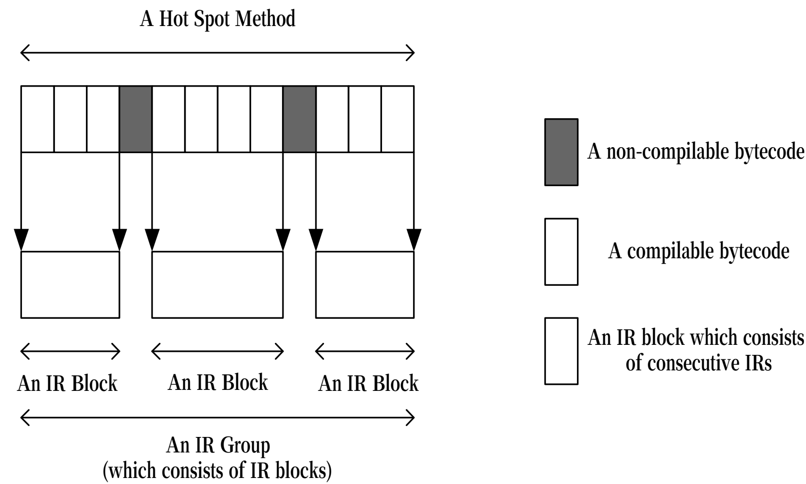


Figure 3-11. Input and Output of the IR Generator

As discussed in Section 3.1, the basic unit of hot spot detection is a method and then it is passed to the IR generator to generate corresponding IR. Figure 3-11 is an illustration that shows the input and the ouput of the IR generator. The detailed workflow is listed as the following steps.

1. The IR generator takes a hotspot method as input.

2. The IR generator linearly parses each bytecode instruction of the method and generates corresponding IR for compile-able bytecode. During the linear pass, the IR generator also updates the PC (program counter) and SP (stack pointer) information for each bytecode instruction. The information is then used by the switching mechanism described in the last paragraph of Section 3.1. For detailed PC and SP offset adjustments of each bytecode instruction, please refer to Appendix A.

3. Consecutively generated IR instructions are collected in a IR block.

4. After IR generation completes, all IR blocks are managed by a IR group. The IR group is then passed to native code generator for code generation.

Since a progam has branch-type instructions, its control flow is not always sequential. In order to overcome this problem, it is necessary to to discover the control structure of the

program by control-flow analysis. However, we reduce the extra cost of control-flow analysis by utilizing the *StackMap* attribute which is specified in the CLDC specification [30]. The *StackMap* attribute records (PC offset, SP offset) tuples for all branch targets in a method. Therefore the IR generator can use the information to identify extended basic blocks. This also implies the maximum range of an IR block is its corresponding extended basic block, provided that there are no intervening non-compile-able bytecode.

3.3.2 Native Code Generator

The main responsibility of the native code generator is to perform register allocation/ assignment and instruction selection/generation. Also some optimizations are applied in this stage.

Since the native code generator is designed for one pass, it implies that register allocation/assignment is done within one pass and instruction selection/generation must be performed at the same time. To be more specific, the native code generator assigns registers as machine instructions are generated. The design of the register allocation/assignment scheme is simple, but highly customized for the JVM environemnt. Its detailed discussion is deferred until Section 3.4.1.

After the IR generation phase, the native code generator receives an IR group as input for code generation. However, the basic unit for code generation is confined to an IR block. In fact, local optimizations in KJITC are all restricted to the range of an IR block. During the code generation for an IR block, the code generator parses each IR instruction and generates corresponding machine instructions, and it is also responsible for generating necessary register load/spill instructions. Besides, the native code generator also incoporates optimizations like constant folding and constant propagation which can help to generate better code.

Upon the end of an IR block, the native code generator must spill registers for live variables. As an optimization technique, the native code generator only spills registers for variables whose memory addresses are below the current stack pointer, since variables above the current stack pointer will not be used again in the stack-based JVM.

Similar to the IR generator, the native code generator collects consecutively generated native code for an IR block in a compiled code block. And all compiled code blocks are managed by a compiled code group. What is worthy of noting is that a compiled code block resided in the compiled code buffer is in reality the basic unit for native execution.

3.4 KJITC Optimizations

We devote this section to the design of major optimization techniques in KJITC. These two optimizations - stack operation folding and rule-based null-pointer check elimination - are designed with the characteristics of the JVM in mind and thus are highly-customized and efficient.

3.4.1 Instruction Folding For Stack Operations

One characteristic of the stack-based JVM is all operations must be done within the Java stack. When mapping the stack-based architecture to the common register-based architecture, this imposes great restrictions and also leads to much inefficiency. Considering the bytecode sequence "*ILOAD_0, ILOAD_1, IADD, ISTORE_0*", its high-level operations are illustrated in Figure 3-12 (a). If these operations can be simplified as shown in Figure 3-12 (b), the execution flow will become more efficient. This technique is called "stack operation folding" in researches on Java processors [31][32].

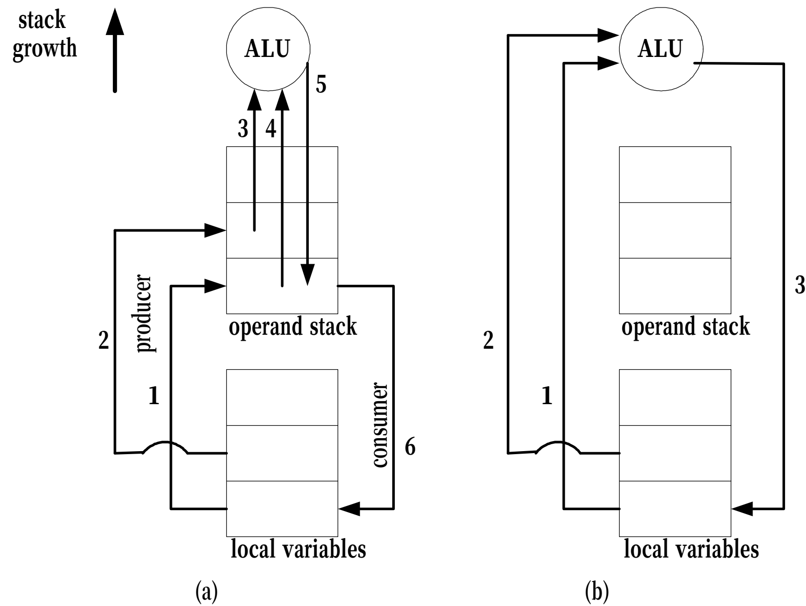


Figure 3-12. Stack Operations (a) Without Folding (b) With Folding

As a contrast, the bytecode sequence can be one-to-one translated into IR instructions as in Figure 3-13 (a). It is observed that the three copy assignments (IR_1, IR_2, and IR_4) can be folded into the third IR instruction by replacing corresponding source and destination fields. After the folding, only one IR instruction is needed instead of four, as in Figure 3-13 (b). This optimization is different from copy propagation in that copy propagation only allows IR_1 and IR_2 to be forward folded into IR_3 while it also allows IR_4 to be backward folded into IR_3.

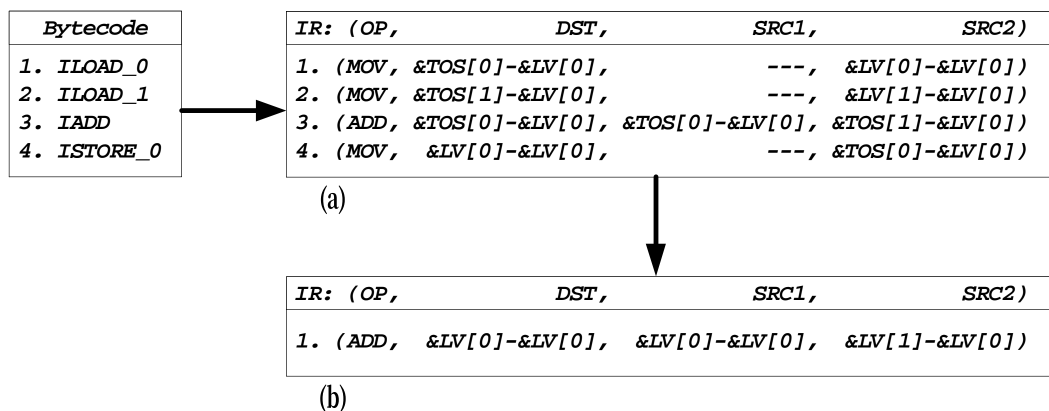


Figure 3-13. IR Generation (a) Without Optimization (b) With Instruction Folding

It is straightforward that the instruction folding technique can be employed in the KJITC by inserting one extra pass between the IR generation and the native code generation. However, devoting one extra pass for only one optimization technique is not cost-effective and also slows down compilation speed. Instead, we integrate this optimization in our native code generator.

The register tracking scheme in our native code generator associates each register record with two two fields - one source and one destination. While encountering a *MOV*-type IR instruction, say the first IR instruction in Figure 3-13 (a), the code generator allocates/assigns a register, and records corresponding source and destination. Later, when the code generator sees the third IR instruction, it will use the allocated/assigned register as the first source. This way, unnecessary stack operations can be effectively removed. Compared with the register allocator in [33], ours is more lightweight and cost-effective. Figure 3-14 is a corresponding work flow of the aforementioned bytecode sequence.

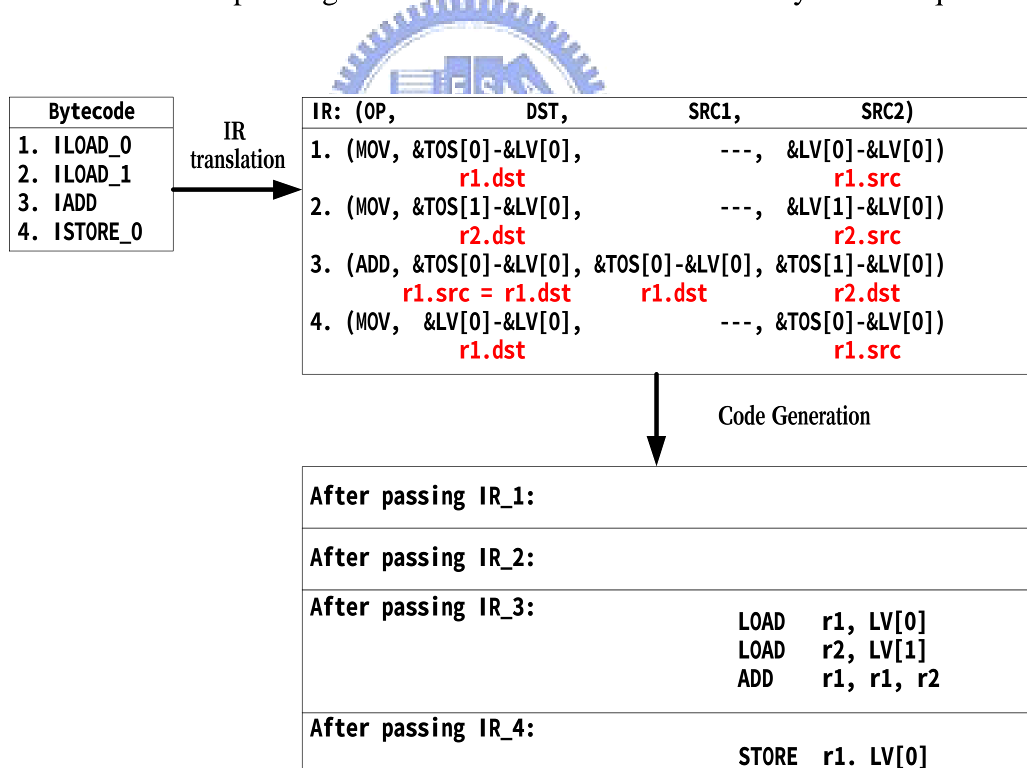


Figure 3-14. Instruction Folding for Stack Operations During Code Generation

3.4.2 Rule-based Null Pointer Check Elimination

Due to its architectural design, the JVM consists of many bytecode instructions that introduce null pointer checks. For example, in KVM "*GETFIELD_FAST*", "*PUTFIELD_FAST*" are for object field access and "*IALOAD*", "*IASTORE*" for array element access, which overall impose much runtime overhead. To reduce such overhead, we propose a rule-based method which is employed in our IR generator. It can eliminate a great portion of IR instructions for null pointer checks in a cost-effective manner, in contrast to other methods employing data-flow analysis.

Now the basic design of the method is described as follows.

- Definition

1. Full Set: (F-Set)

   All compile-able bytecode instructions in the KJITC constitute this set.

2. Un-eliminated Set: (U-Set)

   All bytecode instructions in F-Set, which introduce null pointer checks by examining associated object references, constitute this set.

3. Target Set: (T-Set)

   A predetermined subset of U-Set.

4. Dominance Set: (D-Set)

   All bytecode instructions in F-Set, which produce object references that are later used by bytecode instructions in T-Set, constitute this set.

5. Influential Set: (I-Set)

   All bytecode instructions in F-Set, which may alter object references that are later used by bytecode instructions in T-Set, constitute this set.

- Data Structure

1. A n-height stack (L-Stack)

This is a tiny stack used to simulate stack operations. n poses a limit to the maximum stack height that can be tracked. This stack is implemented as a n-element array.

2. A m-bit-mask array (B-Array)

This array, say array[0:m-1], is used to track whether the local variable 0 through local variable m-1 is null pointer checked or not.

- Algorithm

1. Select some bytecode instructions from U-Set as T-Set

2. Find the corresponding D-Set, I-Set

3. Upon an IR block entrance, initialize all n elements of L-Stack as "Not_Tracked". When some bytecode instruction in D-Set is encountered, mark the corresponding element in L-Stack with the corresponding local variable number.

4. Upon an IR block entrance, initialize all m bits of B-Array as "Un-Checked". When some bytecode instruction in I-Set is encountered, mark the corresponding bit in B-Array with "Un-Checked".

5. When some bytecode instruction in T-Set is encountered, if the bit mask of the local variable associated with the object reference is "Checked", the null pointer check for this bytecode instruction is eliminated; otherwise the null pointer check remains and also the bit mask is then marked as "Checked".

6. The flowchart of the algorithm is depicted in Figure 3-15.

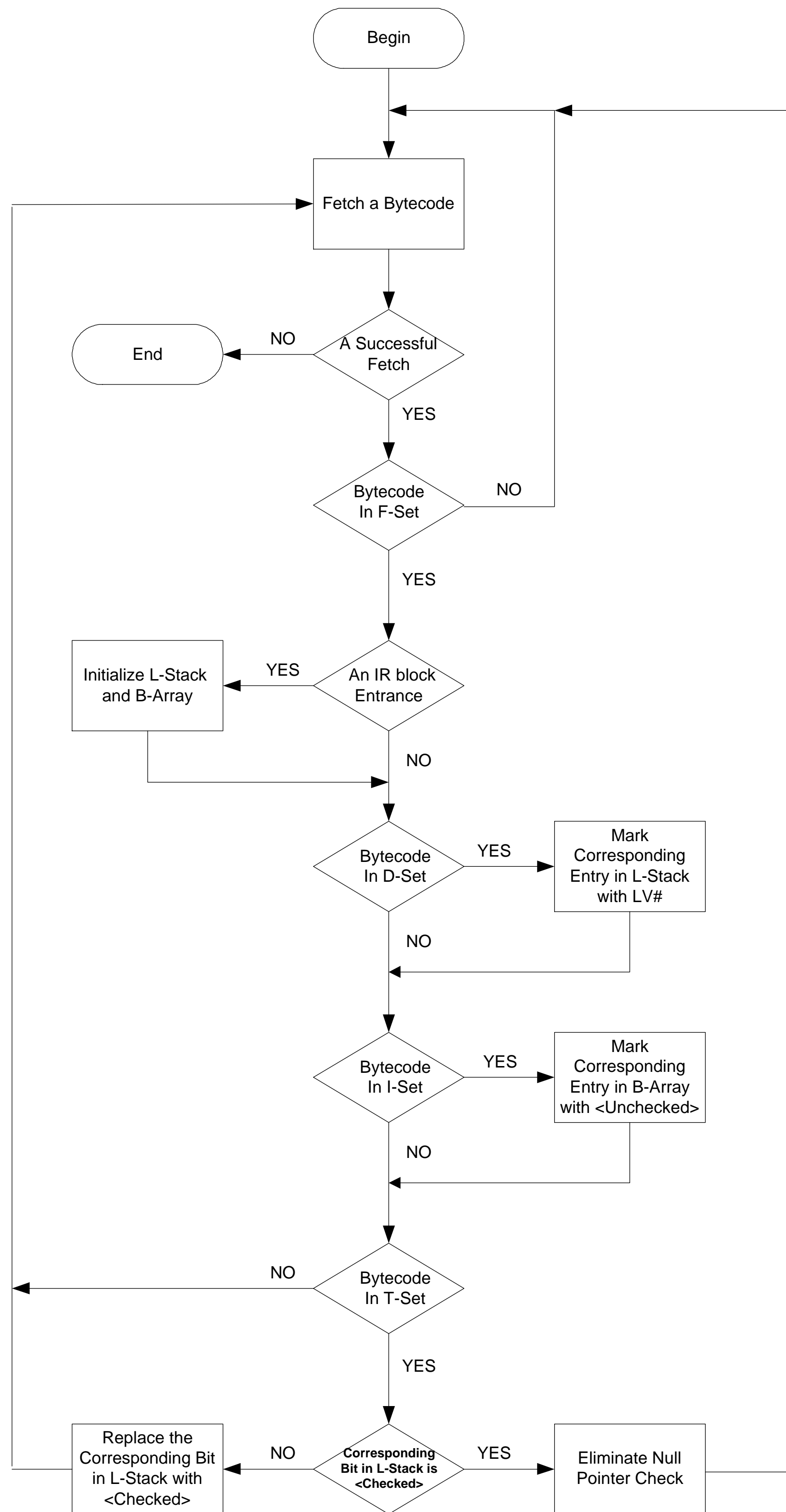


Figure 3-15. Flowchart of Null Pointer Check Elimination

- A Simple Example

1. Configuration:

   T-SET = {"GETFIELDP_FAST", "PUTFIELD_FAST" }

   D-SET = {"ALOAD", "ALOAD_0", "ALOAD_1", "DUP"}

   I-SET = {"ASTORE", "STORE", "ISTORE_0", "ISTORE_1"}

   Pick L-Stack as a 5-height stack

   Pick B-Array as a 4-bit-mask array

2. Given Code Sequence Within an IR-block (see Table 3-1):

   NT: Not Tracked

   NC: Not Checked (or Un-Checked)

   C: Checked

Table 3-1. An Example of Rule-based Null Pointer Check Elimination

Byte-code	Stack Height (after execution)	L-Stack (after execution)	B-Array (after execution)	Perform Null Pointer Check
<Initialize>	0	{NT,NT,NT,NT, NT}	{NC, NC, NC, NC}	
ILOAD xxx	1	{NT,NT,NT,NT, NT}	{NC, NC, NC, NC}	
ALOAD_1	2	{NT, 1, NT, NT, NT}	{NC, NC, NC, NC}	
GETFIELDP_FAST xxx	2	{NT, 1, NT, NT, NT}	{NC, C, NC, NC}	YES
IF_ICMPLT xxx	0	{NT, 1, NT, NT, NT}	{NC, C, NC, NC}	
ALOAD_1	1	{1, 1, NT, NT, NT}	{NC, C, NC, NC}	
ILOAD_2	2	{1, 1, NT, NT, NT}	{NC, C, NC, NC}	
PUTFIELD_FAST xxx	0	{1, 1, NT, NT, NT}	{NC, C, NC, NC}	NO
ICONST_2	1	{1, 1, NT, NT, NT}	{NC, C, NC, NC}	
ISTORE_1	0	{1, 1, NT, NT, NT}	{NC, NC, NC, NC}	
ALOAD_0	1	{0, 1, NT, NT, NT}	{NC, NC, NC, NC}	
GETFIELDP_FAST 0x04	1	{0, 1, NT, NT, NT}	{C, NC, NC,NC}	YES
ICONST_0	2	{0, 1, NT, NT, NT}	{C, NC, NC,NC}	
ICONST_1	3	{0, 1, NT, NT, NT}	{C, NC, NC,NC}	
IASTORE	0	{0, 1, NT, NT, NT}	{C, NC, NC,NC}	YES

In the above example, we observe that the last bytecode, *IASTORE*, also receives a reference as its first input parameter. However, to support bytecode instructions of this type, such as *IALOAD* and *IASTORE*, our basic design needs to be extended somewhat.

Conceptually, an additional stack and an additional bit-mask array must be added to track the field index and the field status respectively, just as L-Stack and B-Array track the local variable number and the local variable status. Detailed description on the algorithm of our extended design is lengthy and therefore is skipped over.

3.5 ARM/Thumb Instruction Set Selection

In order to evaluate the effectiveness of dual instruction set, we choose ARM/Thumb as our target for native code generation. General discussion on ARM and Thumb instruction sets can be found in [6][34]. In this section, we only discuss their differences that relate to our native code generator design.

Register Pressure

In ARM mode, there are 16 registers (R0 ~ R15) available. Excluding R15 (PC: program counter), R14 (LR: link register), and R13 (SP: stack pointer), there are still 13 registers that can be freely used for register allocation/assignment. But since our design involves relative addressing that is local-variable-based (see Section 3.3.1), we devote R0 to storing the starting address of the local variable array. Overall, we have 12 registers left.

In Thumb mode, only 8 registers (R0 ~ R7) can be used without restrictions. Excluding R0, there only remains 7 registers. Therefore, as far as register pressure is concerned, a program compiled in Thumb instruction set will have more load/store instructions for register restoration/spilling than that in ARM.

Instruction Selection

Static compilers in general environment invest much time in instruction selection. Indeed, selecting faster instructions will improve compiled code quality in terms of execution speed. However, due to the demands for fast compilation, our native code generator will only select essential types of instructions for code generation. Here the most important issue is on the width of immedaite field.

In common register-set design, the source fields of instructions may be specified as immediate fields. That is, immediate values or constants within range can be directly encoded in these fields. For those immediate values that exceed the maximum range of the fields, load instructions are needed to retrieve immediate values from memory to registers before these values are used.

Because Thumb instruction set is 16-bit, there is no much space for immediate values when compared with 32-bit ARM instruction set. It seems that insufficient immediate field width may have a great influence on the code quality of the compiled code.

In Table 3-2, we list major types of selected instructions used in our native code generator, thier immediate field widths, their addressing modes (if needed), and thier addressing ranges (if needed). According to the table, differences in ARM/Thumb come from three types of instructions. For detailed explanation, the branch type instruction is used for branching within a method. The PC-relative load/store instruction is used for retrieving constants. The base-addressing load/store instruction is used for accessing the local variables and the operand stack.

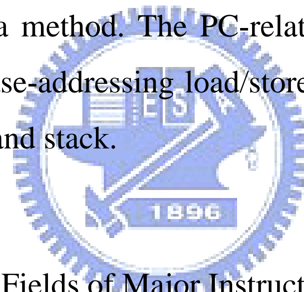


Table 3-2. Immediate Fields of Major Instruction Types

Instructions Type	**Immediate Field In Thumb**	**Immediate Field in ARM**
MOV	imm8	imm8
MUL	N/A	N/A
ADD SUB	imm8	imm8
LSR LSL	imm5	imm5
CMP	imm8	imm8
B	s_imm11 (+-2048 bytes)	imm24 (+-32 Mbytes)
LD (PC-relative) ST (PC-relative)	imm8 (+1024 bytes)	imm12 (+-4096 bytes)
LD (base addressing) ST (base addressing)	imm5 (+128 bytes)	imm12 (+-4096 bytes)

Chapter 4 Experiments

This chapter is devoted to experiments. We first describe our set-up environment for experiments. Next, appropriate benchmarks are chosen for performance evaluation. Finally, experiment results including speed performance and memory usage are exhibited.

4.1 Experiment Environment

Our KJITC is designed and implemented based on version 1.0.4 of Sun's KVM, the reference implementation of J2ME CLDC. For our research usage, the KVM is ported to ARM's ADS1.2, an development environment which includes compiler, assembler, debugger, and instruction set simulator. For compiling Java benchmark programs and KVM's class libraries, the version of the Java compiler adopted is Sun's J2SDK1.4.2_03. For compiling KVM and our KJITC, maximum optimization is specified with -O2 option, and other options remain default. Last but not least, our target architecture is ARM7TDMI, an uncached Harvard architecture which supports both ARM/Thumb instruction sets.

4.2 Benchmarks

Due to the limited APIs that J2ME CLDC specifies, common Java benchmarks can not be applied in our experiment environment. By referring to related academic researches, we choose Embedded CaffeineMark 3.0 [35] for our experiments.

The Embedded CaffeineMark 3.0 uses 6 tests to measure embedded JVM performance in various aspects. Excluding the floating point test which is not supported in CLDC 1.0, the remaining 5 tests are adopted (see Table 4-1).

Table 4-1. Selected Tests of Embedded CaffeineMark 3.0

Name	Brief Description
Sieve	The classic sieve of Eratosthenes finds prime numbers.
Loop	The loop test uses sorting and sequence generation as to measuree compiler optimization of loops.
Logic	Tests the speed with which the virtual machine executes decision-making instructions.
Method	The Method test executes recursive functional calls to see how well the VM handles method calls.
String	String comparison and concatenation.

The original design of Embedded CaffeineMark 3.0 is each test executes for a fixed amount of time, and the reported score is propotional to the number of times the test is executed. There is a problem that the instruction set simulator on which benchmarks run may report inaccurate system timing information to executed benchmarks. It may cause the reported scores float. In order to solve this problem, we modify the 5 tests to make each of time execute for some fixed workload. And therefore we measure the cycle counts of each test for performance evaluation.

4.3 Experiment Results

4.3.1 Effects of KJITC Optimizations

This section is to test the effectiveness of major optimizations employed in our KJITC. The optimizations include instruction folding for stack operations and rule-based null pointer check elimination. Since optimizations are interrelated, it is not possible to precisely break down effects of all optimizations into the sum of each individual optimization. Therefore we measure the speed performance of all optimizations enabled and the speed performance of all but the intended one optimization. Here the embedded JVM is compiled in ARM and the KJITC also targets ARM.

Table 4-2 lists total execution cycles of different optimization setups and of a pure interpreter.

Table 4-2. Execution Cycles of Different Setups

	Interpreter	All But Instruction Folding	All But Null Pointer Check Elimination	All
Sieve	944,892,616	307,787,761	264,466,266	250,364,494
Loop	995,035,635	237,416,508	195,197,888	166,003,256
Logic	984,611,450	829,966,475	812,832,280	812,837,920
String	996,478,059	324,003,198	247,158,210	246,347,558
Method	1,019,476,798	917,606,695	887,267,272	884,642,630
Average	988,098,912	523,356,127	481,384,383	472,039,172

Figure 4-1 shows the speedup of the optimization setups over the pure interpreter, for ease of understanding. The key observation is that instruction folding has more impact than null pointer check elimination. Also to be noted is that due to their program characteristics, logic and method tests exhibit little speed performance improvement.

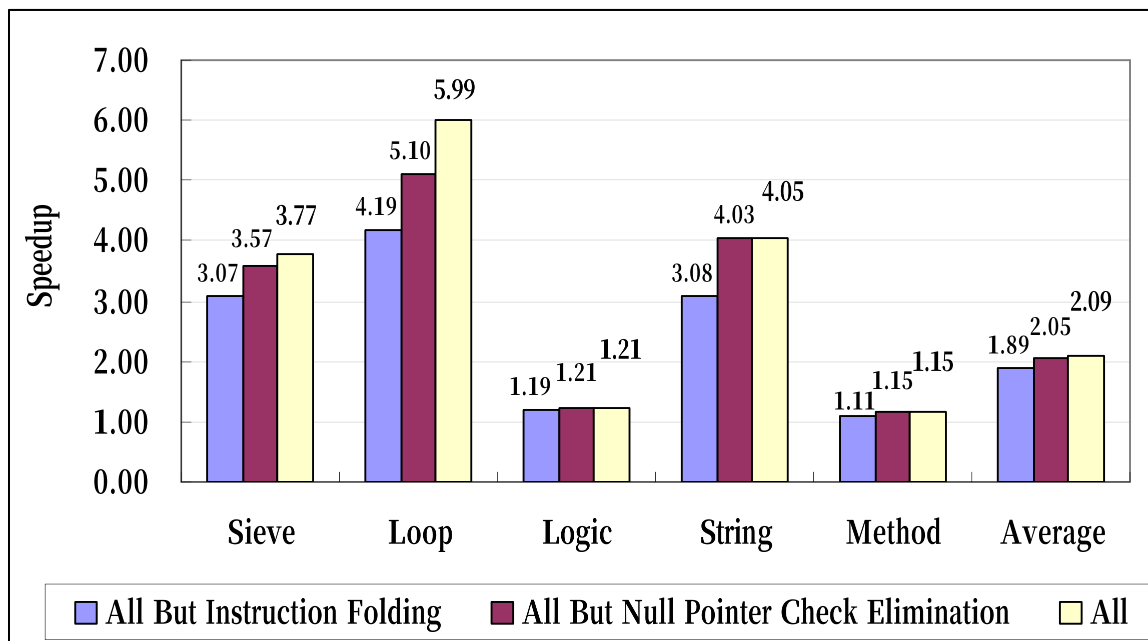


Figure 4-1. Effects of Optimizations

4.3.2 Effects of Dual Instruction Set Selection

To evaluate the impact of dual instruction set selection, we measure the total execution cycles of the following six configurations in Table 4-3. The first two configurations refer to JVMs only with pure interpreters while the last four configurations refer to mixed-mode JVMs that combine interpreters and JIT compilers.

1. Pure Thumb Interpreter (T)
2. Pure ARM Interpreter (A)
3. Thumb JVM + Thumb Compiled Code (T + T)
4. Thumb JVM + ARM Compiled Code (T + A)
5. ARM JVM + Thumb Compiled Code (A + T)
6. ARM JVM + ARM Compiled Code (A + A)

Table 4-3. Execution Cycles of Six Configurations

Benchmarks	Total Cycle Counts (T)	Total Cycle Counts (A)	Total Cycle Counts (T + A)	Total Cycle Counts (T + T)	Total Cycle Counts (A + A)	Total Cycle Counts (A + T)
Sieve	1,463,577,888	944,892,616	276,376,632	279,233,790	250,356,226	253,212,290
Loop	1,567,619,254	995,035,635	171,504,497	174,753,164	165,995,052	169,242,089
Logic	1,553,416,759	984,611,450	1,190,226,734	1,190,223,173	812,846,706	812,842,222
String	1,522,786,128	996,478,059	257,126,996	260,129,554	246,306,962	249,308,107
Method	1,501,301,783	1,019,476,798	1,038,239,460	1,038,237,506	884,634,144	884,631,615
Average	1,521,740,362	988,098,912	586,694,864	588,515,437	472,027,818	473,847,265

Figure 4-2 shows the overall total cycle counts of each configuration versus each test in bar graph. The rightmost test is the average of the five tests. Some of our observations are:

- Pure ARM interpreter is faster than pure Thumb interpreter by about 50%.
- Based on either interpreter (ARM or Thumb), mixed-mode configurations with ARM Compiled Code and Thumb Compiled Code achieve near performance.
- For the Logic test and Method test, the speed improvement of configurations with mixed-mode JVMs over pure interpreters is much less than that for the other three tests.

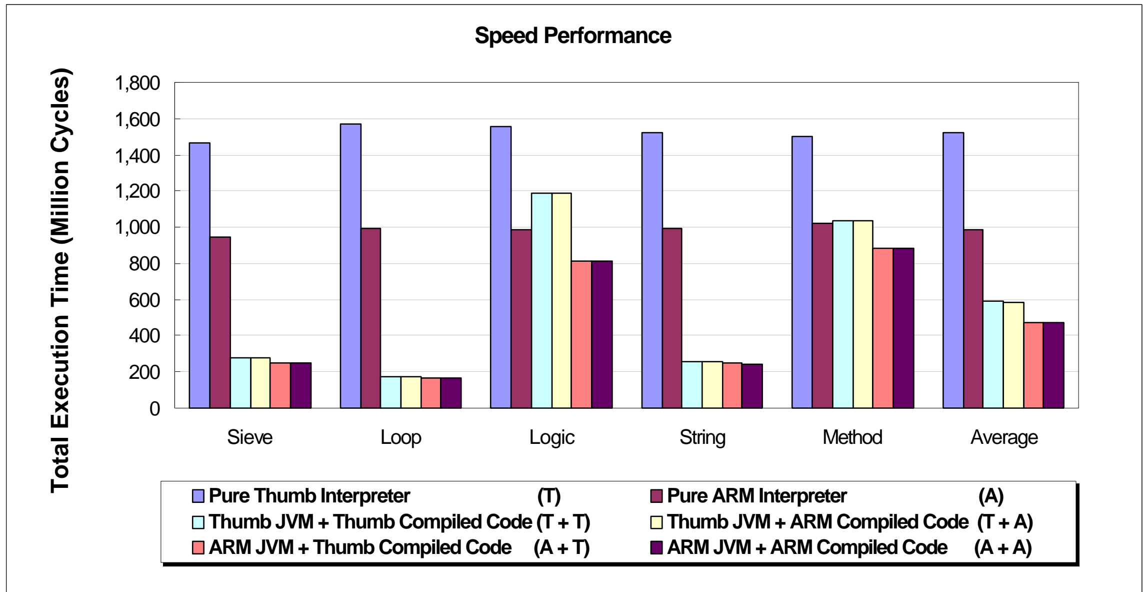


Figure 4-2. Speed Performance of All Configurations

Among the total execution time, JIT compilation time deserves our attention since the cost for compilation is expected to be low enough. For this reason, we normalize compilation cost for a single bytecode as Figure 4-3 depicts. A point to be noted is the compilation cost of the JIT compiler is the sum of the IR generator cost and the corresponding code generator cost.

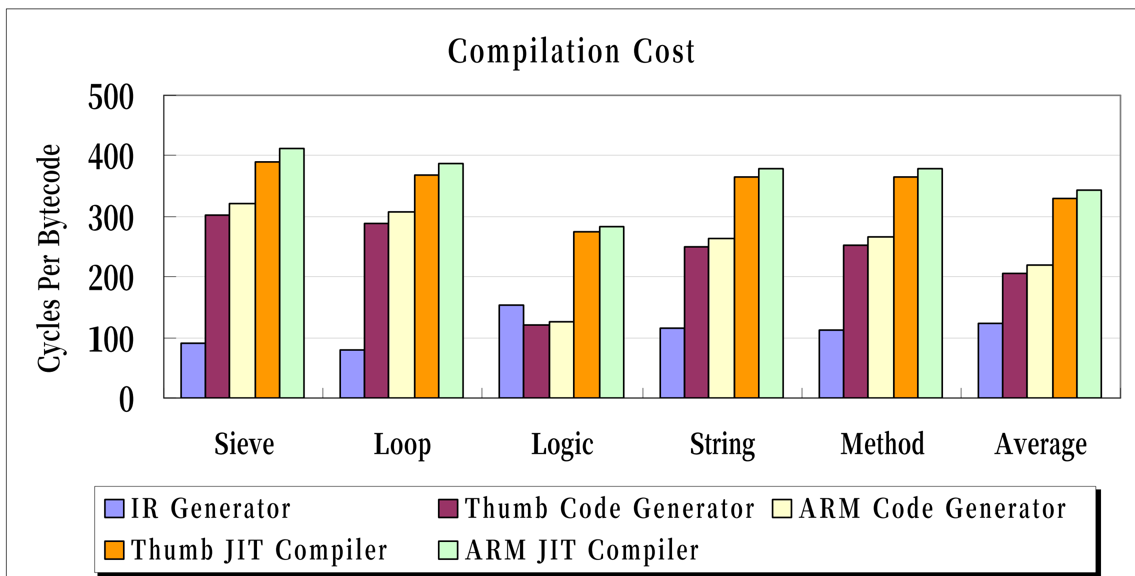


Figure 4-3. Compilation Cost of KJITC

Now we are to consider the static memory usage of each configuration. As shown in Figure 4-4, our ARM JIT compiler take about 23 Kbytes, while Thumb JIT compiler takes about 15 Kbytes. (To be more precise, the four mixed-mode configurations should also consider extra 1~2 Kbytes expansion owing to hot spot detector, switch code, and etc.)

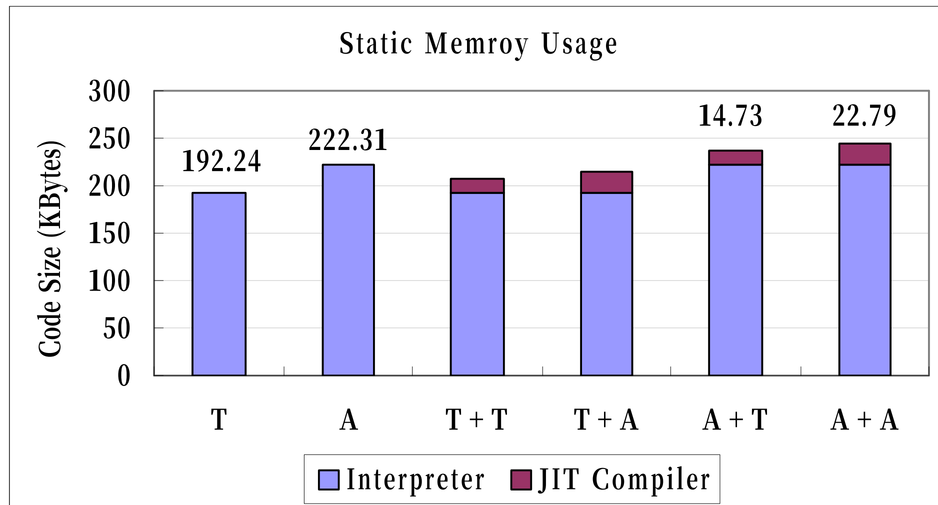


Figure 4-4. Static Memory Usage of All Configurations

Also the dynamic memory usage, by which we mean the compiled code size, of the two JIT compilers is further taken into account. Figure 4-5 demonstrates that the dynamic memory usage of Thumb over ARM is ranged from 60% to 75%, with the average being about 68%.

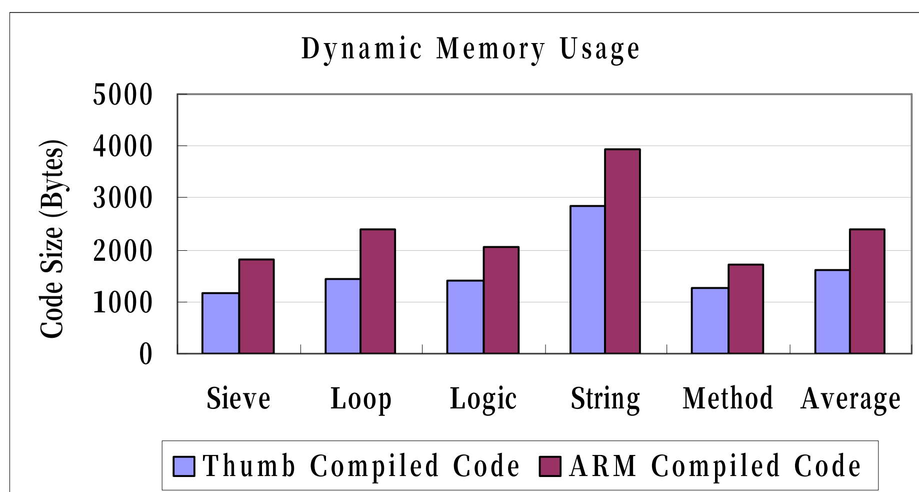


Figure 4-5. Dynamic Memory Usage of the Two JIT Compilers

Finally, in order to evaluate the relative cost-performance of the four mixed-mode configurations, we draw plots of speed increment and total code size increment respectively based on ARM interpreter and on Thumb interpreter (see Figure 4-6). Judged by the two criteria - higher speed increment and lower code size increment, the configuration of ARM interpreter with Thumb JIT compiler seems most cost-effective.

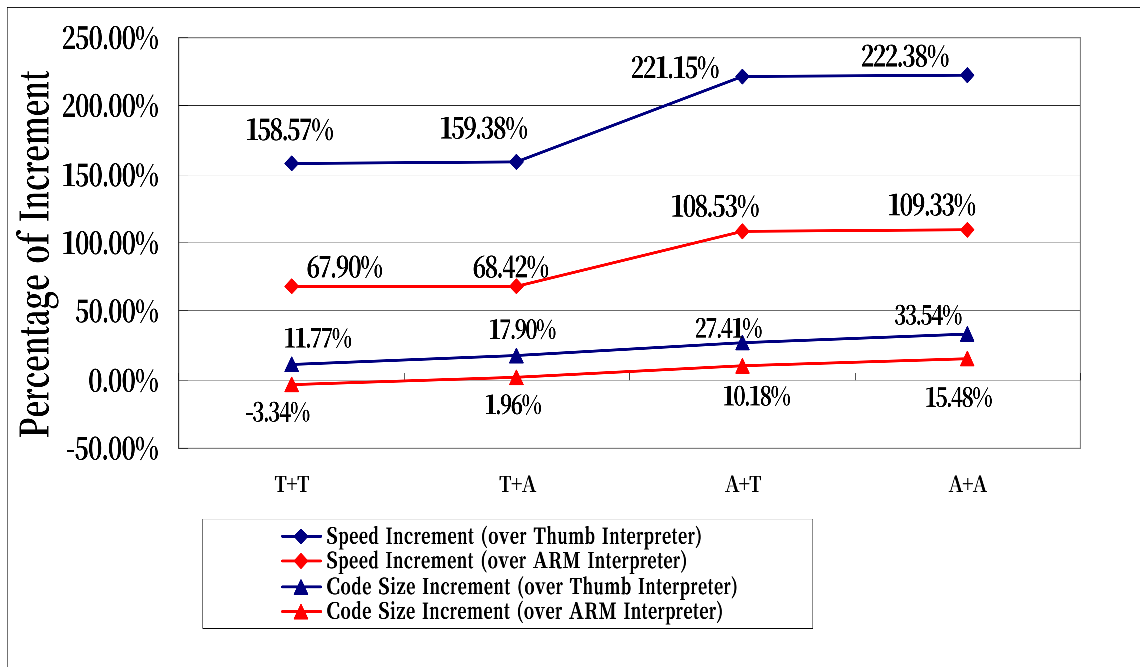


Figure 4-6. Speed Increment and Code Size Increment of Four Mixed-mode Configurations

Chapter 5
Conclusion and Future Work

In this research, our experiment results show that dual instruction set selection has great influence on the performance of an interpreter. The speedup ratio of a pure interpreter-based JVM of ARM over Thumb is about 50%. As far as speed performance is concerned, the ARM interpreter is superior to the Thumb interpreter. However, the case is not hold for the JIT compiler. In fact, the compiled code in ARM and Thumb both exhibit close performance, with ARM compiled code only slightly faster (within 1%) than Thumb compiled code. This result may be quite surprising.

Based on our observation, the Thumb code generator of our JIT compiler only generates less efficient native code than the ARM code generator under either one of the following two conditions. First, there is high register pressure in the Thumb JIT compiler. Second, IR instructions involve immediate values that can not be directly encoded in immediate fields of Thumb instructions. However, since experiment results imply that there is no significant register pressure and the immediate field width is already enough for most cases, we can conclude that static selection of the mixed-mode configuration, the ARM interpreter and the Thumb JIT compiler, is most cost-effective for an embedded JVM. Therefore dual instruction set is statically selected and no dynamic selection is necessary.

This selected configuration on average achieves 2.08 speedup with 10.18% code size increment compared with an ARM interpreter-based JVM, and 3.21 speedup with 27.41% code size increment compared with a Thumb interpreter-based JVM. We also expand one more column from Table 2-1 to compare our KJITC with then others and then remake a new table as Table 5-1.

Table 5-1. Comparison of KJITC with Other JIT Compilers

JIT	Sun - Server	Sun - Client	SNU Latte	Stanford MicroJIT	KJITC
Source	C++	C++	C	C	C
IR Format	SSA dataflow	Simple	Dataflow	Dataflow	Simple
Major Compiler Passes	Interative	4	7	4	2
Register Allocation	Graph coloring	1-pass dynamic	2-pass dynamic	1-pass dynamic	1-pass dynamic
Major Optimizations	1. loop invariant code motion 2. global value numbering 3. constant propagation 4. inlining & specialization 5. instruction scheduling	1. block merging/elimination 2. simple constant propagation 3. inlining & specialization	1. EBB value numbering 2. EBB constant propagation 3. loop invariant code motion 4. inlining & specialization	1. CSE 2. copy propagation 3. constant propagaron 4. loop invariant code motion 5. inlining & specialization 6. instruction scheduling	1. simple constant propagation 2. simple constant folding 3. strength reduction 4. null pointer check elimination 5. instruction folding for stack operations
Compiler Size	1.5MB (Sparc)	700KB (Sparc)	325KB (Sparc)	200KB (Sparc)	15KB (Thumb) 23KB (ARM)
Compilation Cost (Per Bytecode)	~100,000 Cycles	~8,300 Cycles	~20,000 Cycles	~5,000 Cycles	330 Cycles (Thumb) 343 Cycles (ARM)

For future research directions, more optimizations may be incorporated into our KJITC. One optimization that deserves most attention is method inlining. According to our experiment results shown in Figure 4-2, performance improvement on the method test is not significant since there is not much optimization space for tiny method calls in our KJITC. Employing method inlining can be effective under such circumstance.

Study on energy consumption in an embedded mixed-mode JVM is also an interesting research topic. One recent research [36] only discusses energy consumption breakdown of an embedded interpreter-based JVM. As a first thought, our embedded mixed-mode JVM may consume less energy than an interpreter-based JVM for speed performance improvement also leads to energy consumption reduction. However, a more precise method is required to estimate energy consumption of our different configurations that concern both ARM and Thumb.

References

[1] "J2ME Building Blocks for Mobile Devices," Sun Microsystems, May 2000

[2] G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code," *Proc. of USENIX COOTS'97*, 1997

[3] O. Agesen and D. Detlefs, "Mixed-mode Bytecode Execution," TR-2000-87, Sun Microsystems, June 2000

[4] V. Colin de Verdiere, Sebastien Cros, C. Fabre, R. Guider, S. Yovine, "Speedup Prediction for Selective Compilation of Embedded Java Programs," *Proc. of EMSOFT'02*, October 2002

[5] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, A. Nicolau, "A Design Space Exploration Framework for Reduced Bit-width Instruction Set Architecture (rISA) Design," *Proc. of ISSS'02*, October 2002

[6] S. Furber, *ARM System-On-Chip Architecture*, 2nd Edition, Addison Wesley, 2000

[7] D. Sweetman, *See MIPS Run*, Morgan Kaufmann, 1999

[8] B. Venners, *Inside the Java Virtual Machine*, 2nd Edition, McGraw-Hill, 2000

[9] X. Leroy, "Java Bytecode Verification: Algorithm and Formalizations," Journal of Automated Reasoning 30(3-4):235-269, 2003

[10] J. Meyer, T.Downing, *Java Virtual Machine*, O'Reilly, 1997

[11] M. Tremblay, M. O'Connor, "PicoJava: A Hardware Implementation of the Java Virtual Machine," Sun Microsystems, 1996

[12] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, 2nd Edition, Addison Wesley, 1999

[13] W. H. Chiao, *ILP Exploration of Java Stack Operations*, Master Thesis, CSIE, NCTU, 2001

[14] ARM Jazelle Technology, http://www.arm.com/products/solutions/Jazelle.html

[15] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, T. Nakatani, "Overview of the IBM Java Just-In-Time Compiler," IBM Systems Journal, Java Performance Issue, Vol 39, No 1, Februrary 2000

[16] Ali-Reza Adl-Tabatabai, M. Cierniak, G. Y. Lueh, V. M. Parikh, J. M. Stichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler," *Proc. of ACM SIGPLAN'98 Conference on PLDI*, June 1998

[17] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, T. Nakatani, "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler," *Proc. of ACM Java Grande Conference*, June 1999

[18] Alfred V. Aho, Ravi Sethi, Jeffrey d. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1985

[19] Steven. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997

[20] STMicroelectronics, ST100 Technical Manual, http://www.st.com

[21] ARC Cores, ARCtangent-A5 Microprocessor Technical Manual, http://www.arccores.com

[22] A. Krishnaswamy, R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," *Proc. of LCTES'02/SCOPES'02*, June 2002

[23] S. Lee, J. Lee, S. L. Min, J. Hiser, J. W. Davidson, "Code Generation for a Dual Instruction Set Processor Based on Selective Code Transformation," *Proc. of SCOPES'03*, September 2003

[24] D. Gregg, M. A. Ertl, A. Krall, "Implementing an Efficient Java Interpreter," *Proc. of HPCN'01*, June 2001

[25] A. Beatty, K. Casey, D. Gregg, A. Nisbet, "An Optimized Java Interpreter for Connected Devices and Embedded Systems," *Proc. of ACM SAC'03*, March 2003

[26] E. Gagnon, L. Hendren, "Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences," *Proc. of CC'03/ETAPS'03*, January 2003

[27] B. Stephenson, W. Holst, "Multicode: Optimizing Virtual Machines Using Bytecode Sequences," *Proc. of ACM OOPSLA'03*, October 2003

[28] Venugopal K S, G. Manjunath, V. Krishan, "sEc: A Portable Interpreter Optimizing Technique for Embedded Java Virtual Machine," *Proc. of USENIX JVM'02*, August 2002

[29] M. Chen, K. Olukotun, "Targeting Dynamic Compilation for Embedded Environments," *Proc. of USENIX JVM'02*, August 2002

[30] "Connected Limited Device Configuration Specification," Verison 1.1, Sun Microsystems, March 2002

[31] L. R. Ton, L. C. Chang, M. F. Kao, H. M. Tseng, S. S. Shang, R. L. Ma, D. C. Wang, C. P. Chung, "Instruction Folding in Java Processor," *Proc. of ICPADS'97*, December 1997

[32] L. C. Chang, L. R. Ton, M. F. Kao, C. P. Chung, "Stack Operations Folding in Java Processors," *Proc. of IEE Computers and Digital Techniques*, September 1998

[33] C. N. Fischer, R. J. LeBlanc, Jr., *Crafting a Compiler with C*, The Benjamin/Cummings Publishing, 1991

[34] D. Seal, *ARM Architecture Reference Manual*, 2nd Edition, Addison Wesley, December 2000

[35] Pendragon Software Corporation, *Embedded CaffeineMark 3.0 benchmark*, http://www.webfayre.com, 1997

[36] S. Lafond, J. Lilius, "An Opcode Level Energy Consumption Model for a Java Virtual Machine," *Proc. of USENIX VM'04*, May 2004

Appendix
Bytecode Instruction Table

Following is a table that lists all bytecode instructions of our embedded mixed-mode JVM and their short descriptions. It shows whether a bytecode instruction is compiled or not in our current implementation, and corresponding PC and SP offset adjustments. SP offset adjustments can be further expressed in three parts. "Plus" is the number of stack elements the bytecode instruction produces; "Minus" is the number of stack elements the bytecode instruction consumes. "Overall" equals to the value of "Plus" subtracted from "Minus", meaning whether the stack pointer should grow upward or downward. Besides, some offset adjustments can not be determined at static time, and therefore are labeled as "runtime".

It is worth noting that 32 bytecode instructions, from opcode 224 to opcode 255, are originally unused in KVM, but are used in our embedded mixed-mode JVM as marks to identify existing compiled code blocks for native execution. Some fields can not applicable to unused and unneeded bytecode instructions, and are intentionally left black for clarity therefore.

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
0	nop	do nothing	stack manipulation	O	1	0	0	0
1	aconst_null	push null	pushing constants onto the stack	O	1	1	0	1
2	iconst_m1	push int const -1	pushing constants onto the stack	O	1	1	0	1
3	iconst_0	push int const 0	pushing constants onto the stack	O	1	1	0	1
4	iconst_1	push int const 1	pushing constants onto the stack	O	1	1	0	1

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
5	iconst_2	push int const 2	pushing constants onto the stack	O	1	1	0	1
6	iconst_3	push int const 3	pushing constants onto the stack	O	1	1	0	1
7	iconst_4	push int const 4	pushing constants onto the stack	O	1	1	0	1
8	iconst_5	push int const 5	pushing constants onto the stack	O	1	1	0	1
9	lconst_0	push long const 0	pushing constants onto the stack	X	1	2	0	2
10	lconst_1	push long const 1	pushing constants onto the stack	X	1	2	0	2
11	fconst_0	push float 0.0	pushing constants onto the stack	X	1	1	0	1
12	fconst_1	push float 1.0	pushing constants onto the stack	X	1	1	0	1
13	fconst_2	push float 2.0	pushing constants onto the stack	X	1	1	0	1
14	dconst_0	push double 0.0	pushing constants onto the stack	X	1	2	0	2
15	dconst_1	push double 1.0	pushing constants onto the stack	X	1	2	0	2
16	bipush	push byte	pushing constants onto the stack	O	2	1	0	1
17	sipush	push short	pushing constants onto the stack	O	3	1	0	1
18	ldc	push item from runtime constant pool	pushing constants onto the stack	O	2	1	0	1
19	ldc_w	push item from runtime constant pool (wide index)	pushing constants onto the stack	X	3	1	0	1
20	ldc2_w	push long or double from runtime constant pool (wide index)	pushing constants onto the stack	X	3	2	0	2
21	iload	load int from local variable	pushing local variables onto the stack	O	2	1	0	1
22	lload	load long from local variable	pushing local variables onto the stack	X	2	2	0	2
23	fload	load float from local variable	pushing local variables onto the stack	X	2	1	0	1
24	dload	load double from local variable	pushing local variables onto the stack	X	2	2	0	2
25	aload	load reference from local variable	pushing local variables onto the stack	O	2	1	0	1
26	iload_0	load int from local variable 0	pushing local variables onto the stack	O	1	1	0	1
27	iload_1	load int from local variable 1	pushing local variables onto the stack	O	1	1	0	1
28	iload_2	load int from local variable 2	pushing local variables onto the stack	O	1	1	0	1
29	iload_3	load int from local variable 3	pushing local variables onto the stack	O	1	1	0	1
30	lload_0	load long from local variable 0	pushing local variables onto the stack	X	1	2	0	2

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
31	lload_1	load long from local variable 1	pushing local variables onto the stack	X	1	2	0	2
32	lload_2	load long from local variable 2	pushing local variables onto the stack	X	1	2	0	2
33	lload_3	load long from local variable 3	pushing local variables onto the stack	X	1	2	0	2
34	fload_0	load float from local variable 0	pushing local variables onto the stack	X	1	1	0	1
35	fload_1	load float from local variable 1	pushing local variables onto the stack	X	1	1	0	1
36	fload_2	load float from local variable 2	pushing local variables onto the stack	X	1	1	0	1
37	fload_3	load float from local variable 3	pushing local variables onto the stack	X	1	1	0	1
38	dload_0	load double from local variable 0	pushing local variables onto the stack	X	1	2	0	2
39	dload_1	load double from local variable 1	pushing local variables onto the stack	X	1	2	0	2
40	dload_2	load double from local variable 2	pushing local variables onto the stack	X	1	2	0	2
41	dload_3	load double from local variable 3	pushing local variables onto the stack	X	1	2	0	2
42	aload_0	load reference from local variable 0	pushing local variables onto the stack	O	1	1	0	1
43	aload_1	load reference from local variable 1	pushing local variables onto the stack	O	1	1	0	1
44	aload_2	load reference from local variable 2	pushing local variables onto the stack	O	1	1	0	1
45	aload_3	load reference from local variable 3	pushing local variables onto the stack	O	1	1	0	1
46	iaload	load int from array	retrieving values from arrays	O	1	1	2	-1
47	laload	load long from array	retrieving values from arrays	X	1	2	2	0
48	faload	load float from array	retrieving values from arrays	X	1	1	2	-1
49	daload	load double from array	retrieving values from arrays	X	1	2	2	0
50	aaload	load reference from array	retrieving values from arrays	O	1	1	2	-1
51	baload	load byte or boolean from array	retrieving values from arrays	O	1	1	2	-1
52	caload	load char from array	retrieving values from arrays	O	1	1	2	-1
53	saload	load short from array	retrieving values from arrays	O	1	1	2	-1
54	istore	store int into local variable	popping stack values into local variables	O	2	0	1	-1
55	lstore	store long into local variable	popping stack values into local variables	X	2	0	2	-2
56	fstore	store float into local variable	popping stack values into local variables	X	2	0	1	-1

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
57	dstore	store double into local variable	popping stack values into local variables	X	2	0	2	-2
58	astore	store reference into local variable	popping stack values into local variables	O	2	0	1	-1
59	istore_0	store int into local variable 0	popping stack values into local variables	O	1	0	1	-1
60	istore_1	store int into local variable 1	popping stack values into local variables	O	1	0	1	-1
61	istore_2	store int into local variable 2	popping stack values into local variables	O	1	0	1	-1
62	istore_3	store int into local variable 3	popping stack values into local variables	O	1	0	1	-1
63	lstore_0	store long into local variable 0	popping stack values into local variables	X	1	0	2	-2
64	lstore_1	store long into local variable 1	popping stack values into local variables	X	1	0	2	-2
65	lstore_2	store long into local variable 2	popping stack values into local variables	X	1	0	2	-2
66	lstore_3	store long into local variable 3	popping stack values into local variables	X	1	0	2	-2
67	fstore_0	store float into local variable 0	popping stack values into local variables	X	1	0	1	-1
68	fstore_1	store float into local variable 1	popping stack values into local variables	X	1	0	1	-1
69	fstore_2	store float into local variable 2	popping stack values into local variables	X	1	0	1	-1
70	fstore_3	store float into local variable 3	popping stack values into local variables	X	1	0	1	-1
71	dstore_0	store double into local variable 0	popping stack values into local variables	X	1	0	2	-2
72	dstore_1	store double into local variable 1	popping stack values into local variables	X	1	0	2	-2
73	dstore_2	store double into local variable 2	popping stack values into local variables	X	1	0	2	-2
74	dstore_3	store double into local variable 3	popping stack values into local variables	X	1	0	2	-2
75	astore_0	store reference into local variable 0	popping stack values into local variables	O	1	0	1	-1
76	astore_1	store reference into local variable 1	popping stack values into local variables	O	1	0	1	-1
77	astore_2	store reference into local variable 2	popping stack values into local variables	O	1	0	1	-1
78	astore_3	store reference into local variable 3	popping stack values into local variables	O	1	0	1	-1
79	iastore	store into int array	storing values in arrays	O	1	0	3	-3
80	lastore	store into long array	storing values in arrays	X	1	0	4	-4
81	fastore	store into float array	storing values in arrays	X	1	0	3	-3
82	dastore	store into double array	storing values in arrays	X	1	0	4	-4
83	aastore	store into reference array	storing values in arrays	X	1	0	3	-3
84	bastore	store into byte or boolean array	storing values in arrays	X	1	0	3	-3

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
85	castore	store into char array	storing values in arrays	X	1	0	3	-3
86	sastore	store into short array	storing values in arrays	X	1	0	3	-3
87	pop	pop the top operand stack value	stack manipulation	O	1	0	1	-1
88	pop2	pop the top one or two operand stack values	stack manipulation	O	1	0	2	-2
89	dup	duplicate the top operand stack value	stack manipulation	O	1	2	1	1
90	dup_x1	duplicate the top operand stack value and insert two values down	stack manipulation	O	1	3	2	1
91	dup_x2	duplicate the top operand stack value and insert two or three values down	stack manipulation	O	1	4	3	1
92	dup2	duplicate the top one or two operand stack values	stack manipulation	O	1	4	2	2
93	dup2_x1	duplicate the top one or two operand stack values and insert two or three values down	stack manipulation	O	1	5	3	2
94	dup2_x2	duplicate the top one or two operand stack values and insert two, three, or four values down	stack manipulation	O	1	6	4	2
95	swap	swap the top two operand stack values	stack manipulation	O	1	1	1	0
96	iadd	add int	arithmetic	O	1	1	2	-1
97	ladd	add long	arithmetic	X	1	2	4	-2
98	fadd	add float	arithmetic	X	1	1	2	-1
99	dadd	add double	arithmetic	X	1	2	4	-2
100	isub	subtract int	arithmetic	O	1	1	2	-1
101	lsub	subtract long	arithmetic	X	1	2	4	-2
102	fsub	subtract float	arithmetic	X	1	1	2	-1
103	dsub	subtract double	arithmetic	X	1	2	4	-2
104	imul	multiply int	arithmetic	O	1	1	2	-1
105	lmul	multiply long	arithmetic	X	1	2	4	-2
106	fmul	multiply float	arithmetic	X	1	1	2	-1
107	dmul	multiply double	arithmetic	X	1	2	4	-2
108	idiv	divide integer	arithmetic	X	1	1	2	-1
109	ldiv	divide long	arithmetic	X	1	2	4	-2
110	fdiv	divide float	arithmetic	X	1	1	2	-1
111	ddiv	divide double	arithmetic	X	1	2	4	-2
112	irem	remainder int	arithmetic	X	1	1	2	-1
113	lrem	remainder long	arithmetic	X	1	2	4	-2
114	frem	remainder float	arithmetic	X	1	1	2	-1
115	drem	remainder double	arithmetic	X	1	2	4	-2
116	ineg	negate int	arithmetic	X	1	1	1	0

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
117	lneg	negate long	arithmetic	X	1	2	2	0
118	fneg	negate float	arithmetic	X	1	1	1	0
119	dneg	negate double	arithmetic	X	1	2	2	0
120	ishl	shift left int	logical	O	1	1	2	-1
121	lshl	shift left long	logical	X	1	2	3	-1
122	ishr	arithmetic shift right int	logical	O	1	1	2	-1
123	lshr	arithmetic shift right long	logical	X	1	2	3	-1
124	iushr	logical shift right int	logical	O	1	1	2	-1
125	lushr	logical shift right long	logical	X	1	2	3	-1
126	iand	boolean and int	logical	O	1	1	2	-1
127	land	boolean and long	logical	X	1	2	4	-2
128	ior	boolean or int	logical	O	1	1	2	-1
129	lor	boolean or long	logical	X	1	2	4	-2
130	ixor	boolean xor int	logical	O	1	1	2	-1
131	lxor	boolean xor long	logical	X	1	2	4	-2
132	iinc	increment local variable by constant	miscellaneous local varib-ale instructions	O	3	0	0	0
133	i2l	convert int to long	conversions	X	1	2	1	1
134	i2f	convert int to float	conversions	X	1	1	1	0
135	i2d	convert int to double	conversions	X	1	2	1	1
136	l2i	convert long to int	conversions	X	1	1	2	-1
137	l2f	convert long to float	conversions	X	1	1	2	-1
138	l2d	convert long to double	conversions	X	1	2	2	0
139	f2i	convert float to int	conversions	X	1	1	1	0
140	f2l	convert float to long	conversions	X	1	2	1	1
141	f2d	convert float to double	conversions	X	1	2	1	1
142	d2i	convert double to int	conversions	X	1	1	2	-1
143	d2l	convert double to long	conversions	X	1	2	2	0
144	d2f	convert double to float	conversions	X	1	1	2	-1
145	i2b	convert int to byte	conversions	X	1	1	1	0
146	i2c	convert int to char	conversions	X	1	1	1	0
147	i2s	convert int to short	conversions	X	1	1	1	0
148	lcmp	compare long	comparisons	X	1	1	4	-3
149	fcmpl	compare float	comparisons	X	1	1	2	-1
150	fcmpg	compare float	comparisons	X	1	1	2	-1
151	dcmpl	compare double	comparisons	X	1	1	4	-3
152	dcmpg	compare double	comparisons	X	1	1	4	-3
153	ifeq	branch if int comparison with zero succeeds (eq)	conditional branches	O	3	0	1	-1
154	ifne	branch if int comparison with zero succeeds (ne)	conditional branches	O	3	0	1	-1

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
155	iflt	branch if int comparison with zero succeeds (lt)	conditional branches	O	3	0	1	-1
156	ifge	branch if int comparison with zero succeeds (ge)	conditional branches	O	3	0	1	-1
157	ifgt	branch if int comparison with zero succeeds (gt)	conditional branches	O	3	0	1	-1
158	ifle	branch if int comparison with zero succeeds (le)	conditional branches	O	3	0	1	-1
159	if_icmpeq	branch if int comparison succeeds (eq)	conditional branches	O	3	0	2	-2
160	if_icmpne	branch if int comparison succeeds (ne)	conditional branches	O	3	0	2	-2
161	if_icmplt	branch if int comparison succeeds (lt)	conditional branches	O	3	0	2	-2
162	if_icmpge	branch if int comparison succeeds (ge)	conditional branches	O	3	0	2	-2
163	if_icmpgt	branch if int comparison succeeds (gt)	conditional branches	O	3	0	2	-2
164	if_icmple	branch if int comparison succeeds (le)	conditional branches	O	3	0	2	-2
165	if_acmpeq	branch if reference com-parision succeeds (eq)	conditional branches	O	3	0	2	-2
166	if_acmpne	branch if reference com-parision succeeds (neq)	conditional branches	O	3	0	2	-2
167	goto	branch always	unconditional branches and subroutines	O	3	0	0	0
168	jsr	jump subroutine; not needed by KVM	unconditional branches and subroutines	X				
169	ret	return from subroutine; not needed by KVM	unconditional branches and subroutines	X				
170	tableswitch	access jump table by index and jump	table jumping	X	runtime	0	1	-1
171	lookupswitch	access jump table by key match and jump	table jumping	X	runtime	0	1	-1
172	ireturn	return int from method	method return	X	1	0	1	-1
173	lreturn	return long from method	method return	X	1	0	2	-2
174	freturn	return float from method	method return	X	1	0	1	-1
175	dreturn	return double from method	method return	X	1	0	2	-2
176	areturn	return reference from method	method return	X	1	0	1	-1
177	return	return void from method	method return	X	1	0	0	0
178	getstatic	get static field from class	manipulating object fields	X	3	runtime	0	runtime
179	putstatic	set static field in class	manipulating object fields	X	3	0	runtime	runtime
180	getfield	fetch field from object	manipulating object fields	X	3	runtime	0	runtime
181	putfield	set field in object	manipulating object fields	X	3	0	runtime	runtime
182	invokevirtual	invoke instance method; dispatch based on class	method invocation	X	3	runtime	runtime	runtime
183	invokespecial	invoke instance method	method invocation	X	3	runtime	runtime	runtime

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
184	invokestatic	invoke a class (static) method	method invocation	X	3	runtime	runtime	runtime
185	invokeinterface	invoke interface method	method invocation	X	5	runtime	runtime	runtime
186	unused							
187	new	create new object	creating objects	X	3	1	0	1
188	newarray	create new array	creating arrays	X	2	1	1	0
189	anewarray	create new array of reference	creating arrays	X	3	1	1	0
190	arraylength	get length of array	miscellaneous array instructions	X	1	1	1	0
191	athrow	throw exception or error	exceptions	X	1	1	1	0
192	checkcast	check whether object is of given type	miscellaneous object operations	X	3	1	1	0
193	instanceof	determine if object is of given type	miscellaneous object operations	X	3	1	1	0
194	monitorenter	enter monitor for object	monitors	X	1	0	1	-1
195	monitorexit	exit monitor for object	monitors	X	1	0	1	-1
196	wide	extend local variable index by additional bytes	miscellaneous local varibale instructions	X	runtime	runtime	runtime	runtime
197	multianewarray	create new multidimentional array	creating arrays	X	4	1	runtime	runtime
198	ifnull	branch if reference null	conditional branches	O	3	0	1	-1
199	ifnonnull	branch if reference not null	conditional branches	O	3	0	1	-1
200	goto_w	branch always (wide index)	unconditional branches and subroutines	O	5	0	0	0
201	jsr_w	jump subroutine (wide index) ; not needed by KVM	unconditional branches and subroutines	X				
202	breakpoint (reserved)	reserverd opcode for debugging purpose	debugging	X	1	n/a	n/a	n/a
203	getfield_fast	fast version of getfield	manipulating object fields	O	3	1	1	0
204	getfieldp_fast	fast version of getfield	manipulating object fields	O	3	1	1	0
205	getfield2_fast	fast version of getfield	manipulating object fields	X	3	2	1	1
206	putfield_fast	fast version of putfield	manipulating object fields	O	3	0	2	-2
207	putfield2_fast	fast version of putfield	manipulating object fields	X	3	0	3	-3
208	getstatic_fast	fast version of getstatic	manipulating object fields	X	3	1	0	1
209	getstaticp_fast	fast version of getstatic	manipulating object fields	X	3	1	0	1
210	getstatic2_fast	fast version of getstatic	manipulating object fields	X	3	2	0	2
211	putstatic_fast	fast version of putstaic	manipulating object fields	X	3	0	1	-1
212	putstatic2_fast	fast version of putstaic	manipulating object fields	X	3	0	2	-2
213	unused							
214	invokevirtual_fast	fast version of invokevirtual	method invocation	X	3	runtime	runtime	runtime
215	invokespecial_fast	fast version of invokespecial	method invocation	X	3	runtime	runtime	runtime

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
216	invokestatic_fast	fast version of invoke-static	method invocation	X	3	runtime	runtime	runtime
217	invokeinterface_fast	fast version of invokeinterface	method invocation	X	5	runtime	runtime	runtime
218	new_fast	fast version of new	creating objects	X	3	1	0	1
219	anewarray_fast	fast version of anewarray	creating arrays	X	3	1	1	0
220	multianewarray_fast	fast version of multi-anewarray	creating arrays	X	4	1	runtime	runtime
221	checkcast_fast	fast version of checkcast	miscellaneous object operations	X	3	1	1	0
222	instanceof_fast	fast version of instanceof	miscellaneous object operations	X	3	1	1	0
(if.) 223	customcode	for special usage	special usage	X	1	n/a	n/a	n/a
224	unused	JIT_SWITCH0						
225	unused	JIT_SWITCH1						
226	unused	JIT_SWITCH2						
227	unused	JIT_SWITCH3						
228	unused	JIT_SWITCH4						
229	unused	JIT_SWITCH5						
230	unused	JIT_SWITCH6						
231	unused	JIT_SWITCH7						
232	unused	JIT_SWITCH8						
233	unused	JIT_SWITCH9						
234	unused	JIT_SWITCH10						
235	unused	JIT_SWITCH11						
236	unused	JIT_SWITCH12						
237	unused	JIT_SWITCH13						
238	unused	JIT_SWITCH14						
239	unused	JIT_SWITCH15						
240	unused	JIT_SWITCH16						
241	unused	JIT_SWITCH17						
242	unused	JIT_SWITCH18						
243	unused	JIT_SWITCH19						
244	unused	JIT_SWITCH20						
245	unused	JIT_SWITCH21						
246	unused	JIT_SWITCH22						
247	unused	JIT_SWITCH23						
248	unused	JIT_SWITCH24						
249	unused	JIT_SWITCH25						
250	unused	JIT_SWITCH26						
251	unused	JIT_SWITCH27						
252	unused	JIT_SWITCH28						

Opcode	Mnemonic	Description	Function Group	Compiled ?	PC offset (8-bit)	SP offset (32-bit)		
						Plus	Minus	Overall
253	unused	JIT_SWITCH29						
254	unused	JIT_SWITCH30						
255	unused	JIT_SWITCH31						