國立交通大學

資訊科學與工程研究所

碩 士 論 文

32bit-16bit 混合指令集嵌入式系統
程式碼減量爪哇即時編譯器

Reducing Code Size in Java JIT Compilers
for 32bit-16bit Mixed Instruction Set Architectures

研 究 生：呂禮君

指導教授：楊　武　博士

中 華 民 國　九 十 七　年 六 月

32bit-16bit 混合指令集嵌入式系統程式碼減量爪哇即時編譯器

Reducing Code Size in Java JIT Compilers for 32bit-16bit Mixed Instruction Set Architectures

研 究 生：呂禮君　　Student：Li-Jyun Lyu

指導教授：楊　武 博士　　Advisor：Dr. Wuu Yang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年六月

32bit-16bit 混合指令集嵌入式系統程式碼減量爪哇即時編譯器

學生：呂禮君　　指導教授：楊　武 博士

國立交通大學資訊科學與工程所碩士班

摘 要

隨著近年來嵌入式系統的市場快速蓬勃，嵌入式系統處理器的執行速度以相當快的速度成長，在處理器的速度越來越快的情況下，機器執行程式的瓶頸，已經由原先處理器的執行速度漸漸轉移到的與處理器與週邊儲存設備溝通的速度，這主要來自於傳輸資料的速度與處理資料的速度落差所造成的處理器空轉的情況。為使程式效能有效提昇，減少記憶體存取次數，以提昇快取成功的機率，在效能提昇上成為一個可行且明確的方法。在嵌入式系統所使用的語言中，爪哇程式語言基於跨平台的特性在嵌入式平台上，一直佔著重要的地位，而跨平台所不得不付出的成本為效能上的低落，為解決此問題，將爪哇語言的 byte code 轉換成平台專屬的 machine code，藉以提昇效能的爪哇即時編譯器，為針對嵌入式平台提昇爪哇程式語言執行效能的最佳解決方案。本篇論文修改爪哇即時編譯器，使爪哇即時編譯器所產生的 machine code，能夠混合產生 32bit-16bit 指令，藉以有效減少程式碼的大小，以降低執行時指令快取失敗的機會，進而減少存取記憶體的次數，以取得效能上的提昇。此外，為了更有效的漸少程式碼，我們運行了一連串的實驗，針對 VM 處理器的特殊指定暫存器配置，及 Register Set 的設定調整。透過這些實驗，可以取得各種配置所能得到的程式碼減量情形，並探討減量所帶來的優化效應，如程式碼大小與程式執行效率之間的變化關西。實驗結果顯示，我們的方法平均可以減少百分之十左右的程式碼大小，同時

幾乎沒有對效能造成負擔，當執行較大型的程式時，甚至能夠達到提昇效能的目的。

關鍵字：爪哇即時編譯器、程式碼減量、32bit-16bit 混合固定長度指令集架構

Reducing Code Size in Java JIT Compilers for 32bit-16bit Mixed Instruction Set Architectures

Student: Li-Jyun Lyu Advisor: Dr. Wuu Yang

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

In recent years, because the market of embedded systems develops quickly, the process speed of embedded systems had rapidly grown. As the processors become faster and faster, the bottleneck of program execution shifts to the communication between CPU and the main memory. The main reason is the increasing gap between CPU speed and memory speed. Reducing code size may potentially reduce the number of memory accesses (by increasing cache hit ratio) and becomes an effective method to improve CPU performance. For this reason, new CPU architectures provide both 16-bit and 32-bit instructions. We developed a new method that can generate a mixture of 16-bit and 32-bit instructions. This method is implemented and tested in a Java just-in-time compiler of a Java virtual machine for the Andes platform. Our experiment shows that the code size can be reduced 10% at very little extra overhead (only 0.14%). The performance improvement for a long-running program can be quite significant.

Keywords: JIT compiler, reduce code size, 32bit-16bit Mixed Instruction Set Architectures

ACKNOWLEDGEMENTS

我誠摯的感謝我的指導教授楊武博士在這兩年的研究時光，不斷的指導我的研究，協助我發展完成這篇論文。老師熱情的教學熱和充分的耐心，不單單幫助我在學業上的進步，同時令我個人全方面的成長。同時感謝產學合作計畫的徐慰中教授及單智君教授在研究方面的指引，老師充滿建設性的建議和討論，令我的論文大幅增加它的價值。感謝口試委員楊朝棟教授在論文上的建議，教授們對研究嚴謹的態度是我學習的榜樣。

另外感謝陳裕生學長在論文發展過程中的指導及程式上的協助，由於學長的指導使我的論文更加完美。感謝沈柏曄學長，在實驗上的協助及指導，令我在實驗的過程中順利的完成收集資料。同時感謝「程式語言與系統實驗室」的同學、顏子軒同學及蔡雙圓同學在這段時間的砥礪，有建設性的討論、建議和幫助。

最後，我要感謝我的父母及姊姊的支持，關心和照顧，如果我能有一點點微小的成就，都是來自於他們，謹將這篇論文獻給我心愛的家人。

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

In recent years, because the market of embedded systems develops quickly, the speed of embedded processor had rapidly grown. The speed of the processor is faster and faster, the bottleneck of program execution switches from processor speed to I/O speed, that is, the speed with a peripheral equipment communicates with a central processor. The main reason is that the difference in speed between transfer of data and handle of data that make processor idle. So reducing code size to decrease cache misses becomes an attractive approach to improve overall performance. We present a JIT compiler for processors with multiple fixed-width instructions. It will effectively reduce code size without undue overhead and will improve the performance of the generated code.

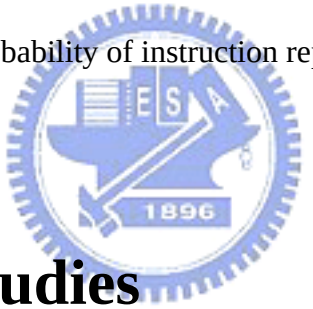
1.1 Motivation

Because embedded systems have many different instruction-set architectures (ISA), the Java language[1][2] becomes important for embedded systems due to its platform independence. But, platform-independence comes with serious performance penalty. To mitigate the performance penalty, Java VM proposes the *just-in-time compiler* architecture[3][4][5], which executes the target-machine code directly for improving performance.

Some RISC processors, such as ARM[6][7], MIPS[8] and ANDES[9][10], support the 32-bit/16-bit multiple fixed-width instruction sets. In this domain, the 16-bit ISA is usually targeted at reduced code size and lower power consumption.

Some of these RISC processors, such as MIPS, requires a mode-switching instruction to switch between the 16-bit and 32-bit modes. This results in overhead at run time. On the other hand, some other processors, such as ANDES, do not need the mode-switch instructions. 16-bit and 32-bit instructions can mix freely in the program.

We present a new code generator for a mixed-instruction JIT in this thesis. Our aim is to reduce the size and to improve the performance of the generated code. Furthermore, the code generator itself is quite efficient. Our target is ANDES 32-bit/16-bit ISA, which has the following features: (1) there is no mode-change instructions; (2) the operation of the 16-bit instructions almost reflective to 32-bit instructions. Almost every 32-bit instruction can be mapped by a suitable 16-bit instruction. It increases the probability of instruction replacement.



1.2 Related Studies

In the domain of reduce code size, using 16-bit instruction set like Thumb must use mode change instruction to mix instruction between 32-bit instruction and 16-bit instruction and has some performance cost. Another disadvantage for the 16-bit instruction set is that fewer registers are available in the 16-bit mode. This will add additional Load/Store instructions.. To balance code size and performance, Lee, S. and Lee, J. proposed a method which is first compile code to 16-bit instruction set. Next, a selected subset of basic block are compiled to 32-bit instruction set. They profile or WCET(worst-case execution time) analysis to decide mixed instruction code.[11-18] Their method incurs much compilation time. For this reason, it is not suitable for a JIT compiler.

1.3 Propose Approach

There are two issues in our study. First, a complex method is not acceptable because it will incur much compilation time, which is part of the total running time. Second, compiling with the 16-bit instruction set will generate more instructions than with the 32-bit instruction set. (However, a 16-bit instruction is only a half (in size) of a 32-bit instruction.) Increasing the number of instructions will decrease the overall performance. We analyze compiled 32-bit code in our benchmarks. We find a lot of opportunities for translating 32-bit instructions into 16-bit counterparts. To be more precise, almost 80% instructions can be translated into 16-bit equivalents. This observation motivates us to propose an efficient method to generate mixed code. Figure 1.1 is Analysis of translatable instruction. Figure 1.2 is Generating mixed code method

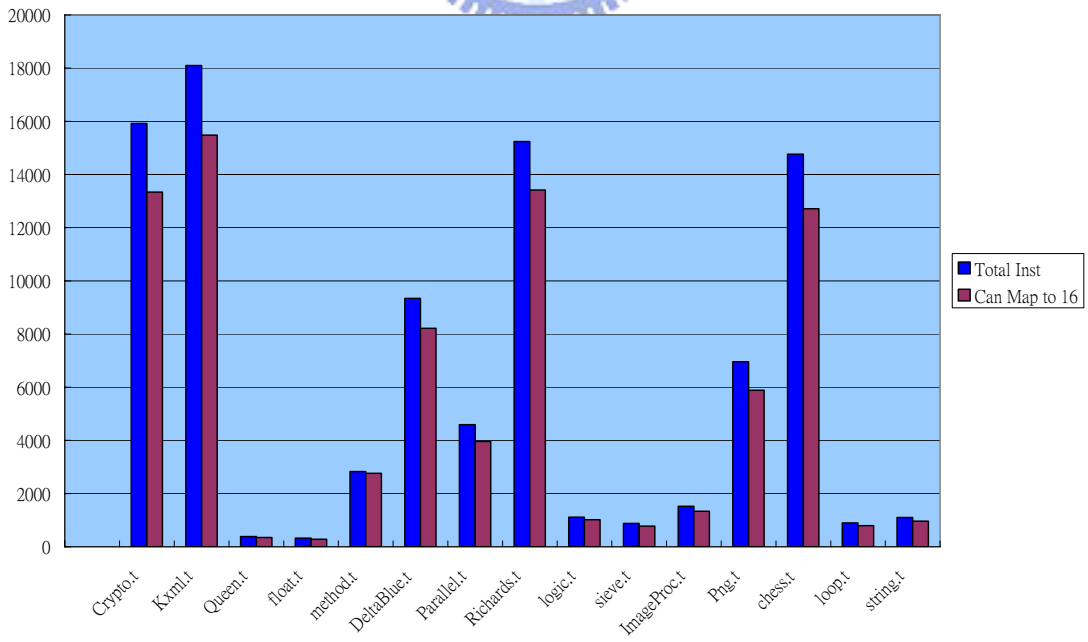


Figure 1.1 Analysis of translatable instruction.

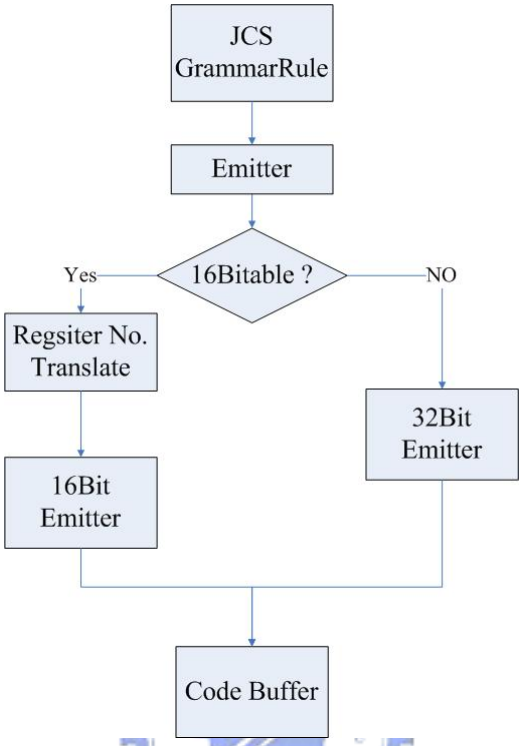


Figure 1.2 Generating mixed code method

1.4 Contribution

Our experiments show that this scheme is successful to reduce code size without too mach overhead. In addition, performance is improved. In the embedded domain, balance between memory size and performance are discussed. This paper presents a novel approach two points: the performance and code size which use ANDES 16-bit ISA.

1.5 Synopsis

The remainder of this thesis is organized as follows. Chapter 2 discusses Java Just-In-Time compiler and Andes 32bit-16bit Instruction Set Architectures. In Chapter 3, we introduce the *Multiple Fixed-width ISA Emitter*. In Chapter 4, experiments and the results are presented and be analyzed. In Chapter 5, the conclusion and future work are given.

Chapter 2
Java Just-In-Time compiler and Andes 32bit-16bit Instruction Set Architectures

As the Java language becomes more and more important for programming embedded systems, translation at the byte-code level has been proposed to increase program performance. Java VM proposes a just-in-time compiler architecture, which executes target-machine code for improve performance.

ANDES ISA proposes a special architecture which uses mixed-mode instructions without the need of mode-switching instructions. Its 16-bit ISA almost reflects to 32-bit ISA, but it has an alignment restriction: 32-bit memory instruction reference object that must be word-alignment.

In our research, our target is the ANDES processor. We port an existing JVM JIT to the ANDES platform and then modify the code emitter so that it can generate16-bit as well as 32-bit instructions. We use several benchmark tests to measure the performance of the code emitter.

2.1 CVM Internals

The virtual machine we use the *Connected Device Configuration Hotspot Implementation* (*CVM*) version of JAVA VM, which is highly optimized for resource-constrained devices, such as consumer electronic products and embedded

devices. Portability is the most important benefits of the Java system design. It includes a dynamic compiler, which is also called a just-in-time compiler (JIT). While a method in the Java program has been used frequently enough, JIT converts the method's bytecodes to native code during execution time to improve future performance. This operation has two passes: First, the front end converts Java bytecode to an intermediate representation (IR); Second, the back end converts the IR to native code. The architecture is shown in Figure 2.1.

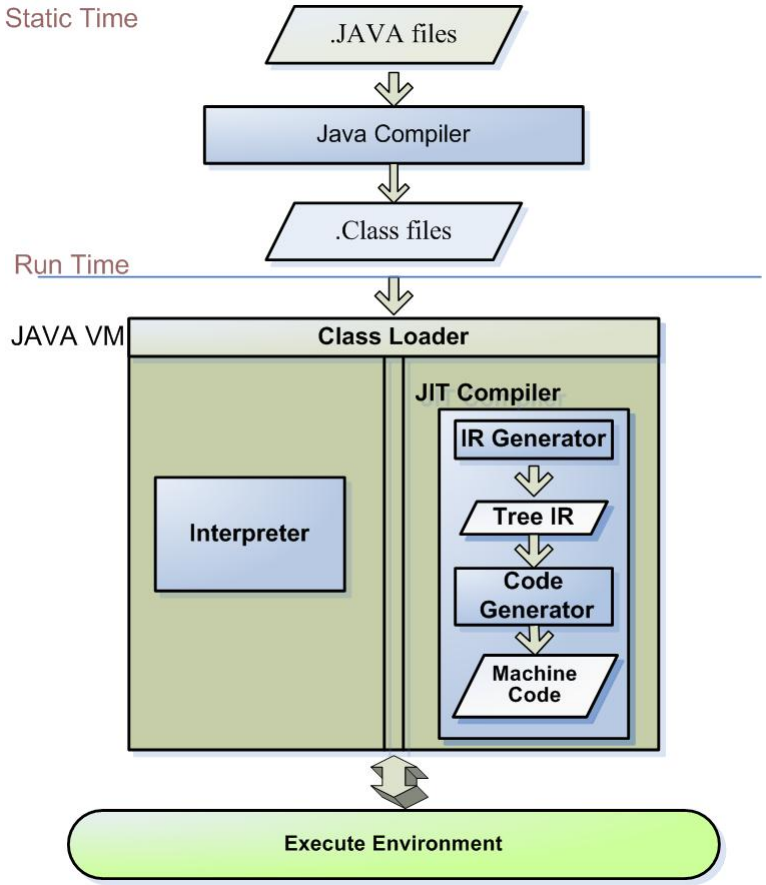


Figure 2.1 Java program execute

2.1.1 JIT Front End

The front end is portable for different execution environments. It converts the bytecode to an intermediate representation (IR). Figure 2.3 is an example of IR.

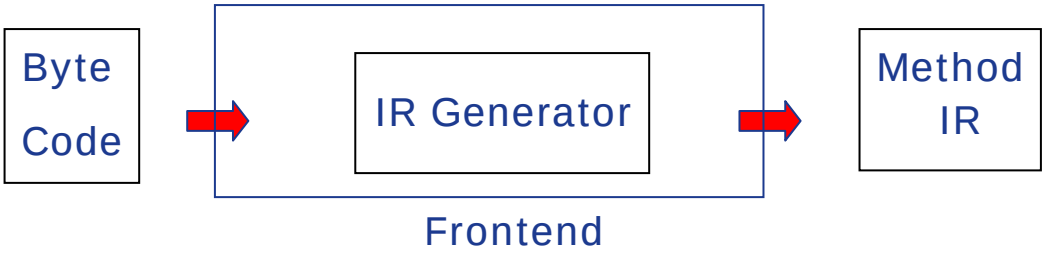


Figure 2.2 Frontend

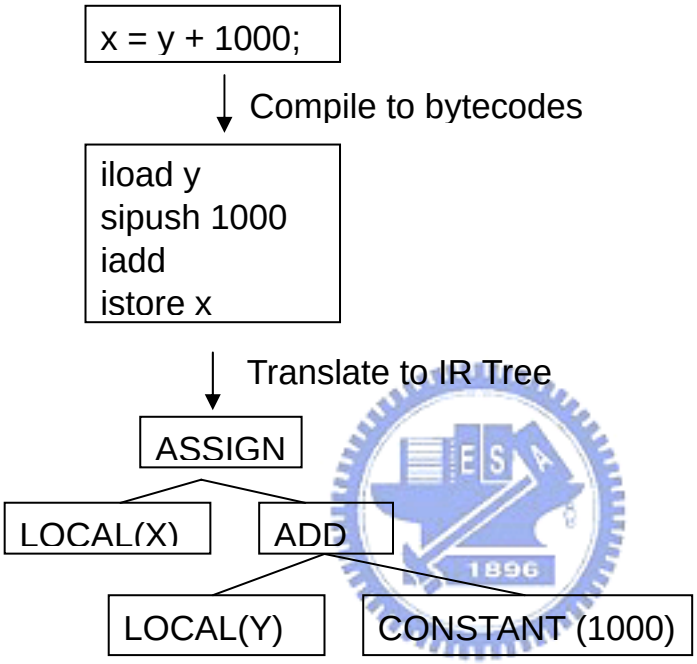


Figure 2.3 An example of IR.

2.1.2 JIT Back End

The back end converts IR to native instructions. An IR tree is parsed by a parser. The parser, which is produced by the *Java Code Select* (JCS) tool at build time, performs pattern matching for tree-based data structures in which the patterns are specified as a set of JCS rules. These rules are translated into C source code and initialized data structures. Code generation is done with rule-based pattern matching on trees. When there are multiple possibilities, JCS choose the rules with the least

static costs. Figure 2.5(A) is an example of JCS rules.

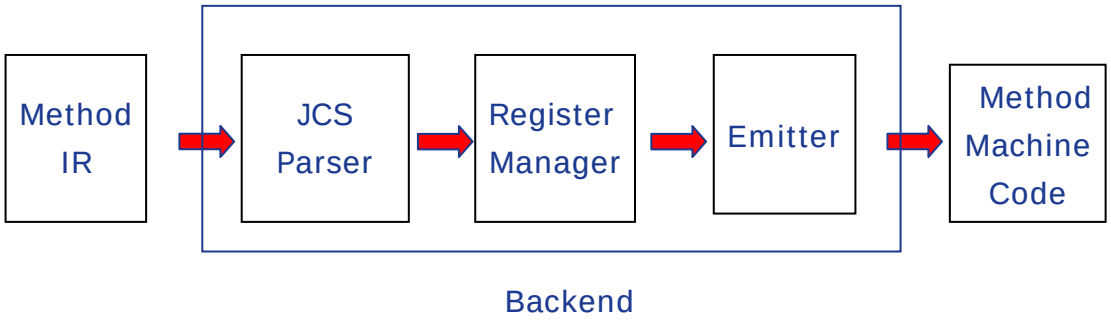


Figure 2.4 Backend

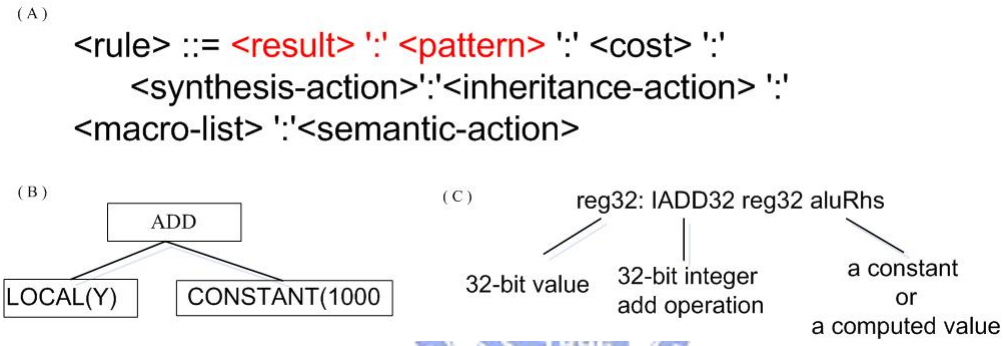


Figure 2.5 An example of a JCS rule

The first part of this rule is the result and the second part is a pattern. They are used for pattern matching. For instance, the subtree in Figure 2.5 (B) will be matched by the JCS rule in Figure 2.5 (C). If a subtree can be matched in multiple ways, the rule with the lowest static cost will be selected. The static cost is specified as the third part of a rule. After a match is found, the fourth and the fifth parts of the rule will be used for setting up a register set. This is shown in Figure 2.6(A). First, a bottom-up traversal of the matched tree passes the use register set, shown in Figure 2.6(B). Second, a top-down traversal passes the accept register set, shown in Figure 2.6(C). After these two passes, the register manager knows which registers are provided. Finally, the last part is the semantic actions which will call the code emitter to emit native instructions. It is shown in Figure 2.7(A) and Figure 2.7(B).

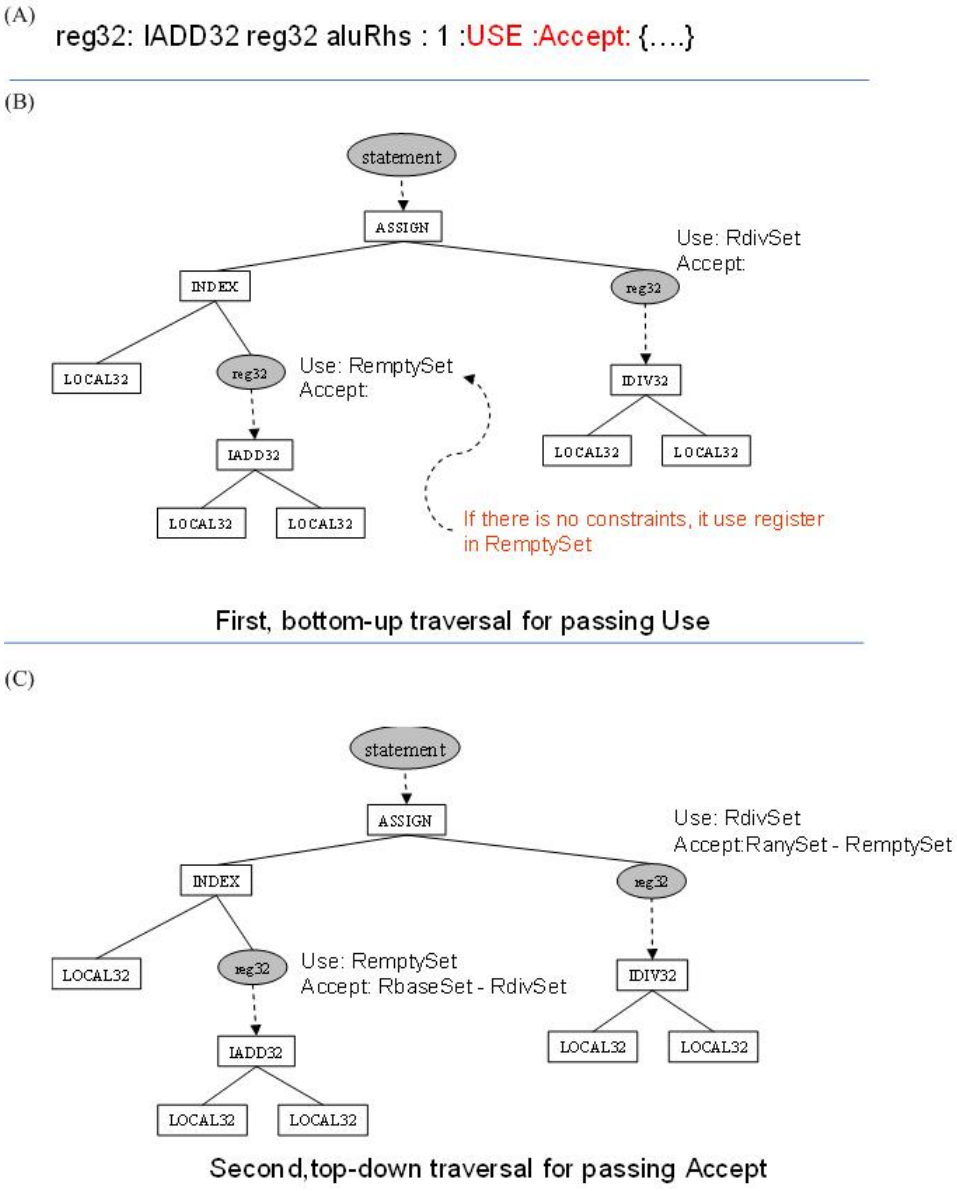


Figure 2.6 Set Register Set

2.2 ANDES Instruction Set Architectures

In our system, we use the ANDES instruction set, which is a RISC-style register-based instruction set. In Andes ISA, we may freely mix 16-bit and 32-bit instructions without the need of mode-switching instructions. The 16-bit ISA almost reflects the 32-bit ISA, but there is an alignment restriction: When a 32-bit

instruction is written to the code buffer, the address of the memory cell in the code buffer that will hold the instruction must be word-aligned. Otherwise, the 32-bit instruction must be broken into two 16-bit halves. Each half is written to the code buffer separately. A Word-Alignment exception will be thrown when we attempt to write a 32-bit instruction at half-word alignment.

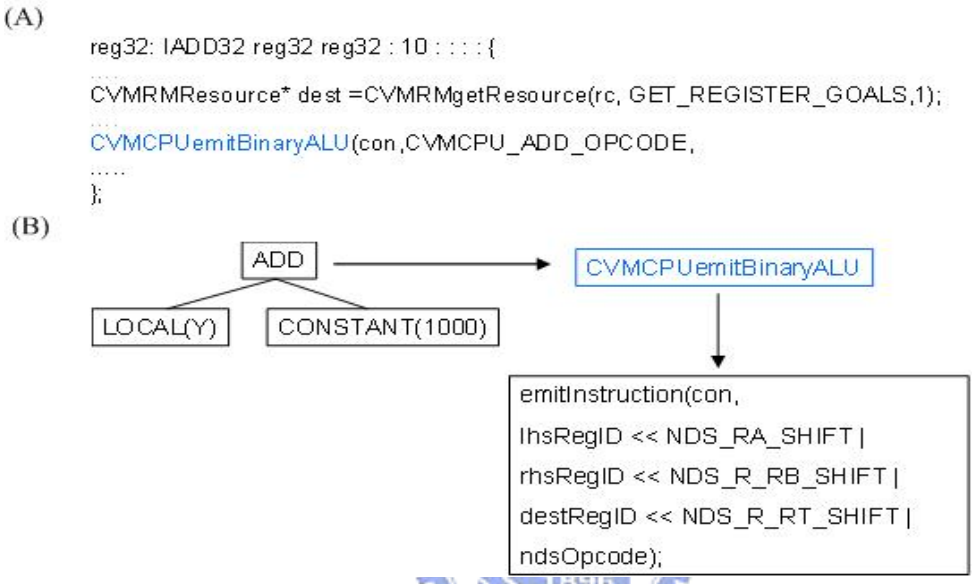


Figure 2.7 JCS rule calls emitter to emit native code

2.2.1 General Purpose Register

Andes 32-bit instructions can access thirty-two 32-bit general-purpose registers (GPR). A 16-bit instruction's register index can be 5 bits, 4 bits, or 3 bits in different instruction formats. A 3-bit and 4-bit index can only access a part of the GPRs. The 3-bit and 4-bit register indices are mapped to real registers according to Table 2.1.

2.2.2 The Andes Instruction Set

In this section, we introduce the part of the Andes instruction set that is related to our research. In Andes, the memory address accessed by a 32-bit memory instruction has to be word-aligned. Otherwise, a Data Alignment Check exception will be

generated. Table 2.2 - 2.8 are examples of which the maps of 32-bit instruction translate to 16 bit instruction.

Table 2.1 Andes General Purpose Registers

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
R0	A0	H0	O0	
R1	A1	H1	O1	
R2	A2	H2	O2	
R3	A3	H3	O3	
R4	A4	H4	O4	
R5	A5	H5	O5	Implied register for beqs38 and bnes38
R6	S0	H6	O6	Saved by callee
R7	S1	H7	O7	Saved by callee
R8	S2	H8		Saved by callee
R9	S3	H9		Saved by callee
R10	S4	H10		Saved by callee
R11	S5	H11		Saved by callee
R12	S6			Saved by callee
R13	S7			Saved by callee
R14	S8			Saved by callee
R15	Ta			Temporary register for assembler Implied register for slt(s|i)45, b[eq|ne]zs8
R16	T0	H12		Saved by caller
R17	T1	H13		Saved by caller
R18	T2	H14		Saved by caller
R19	T3	H15		Saved by caller
R20	T4			Saved by caller
R21	T5			Saved by caller
R22	T6			Saved by caller
R23	T7			Saved by caller
R24	T8			Saved by caller
R25	T9			Saved by caller
R26	P0			Reserved for Privileged-mode use.
R27	P1			Reserved for Privileged-mode use.
R28	S9/Fp			Frame pointer / Saved by callee
R29	Gp			Global pointer
R30	Lp			Link pointer
R31	Sp			Stack pointer

Table 2.2 Add/Sub Instruction

32-bit instruction	16-bit instruction	Special case
ADD	ADD333	
	ADD45	
SUB	SUB333	
	SUB45	
ADDI	ADDI333	
	ADDI45	
	SUBI333	
	SUBI45	

Table 2.3 Move instruction

32-bit instruction	16-bit instruction	Special case
MOVI	MOVI55	
ADDI/ORI	MOV55	ADDI R# R# 0

Table 2.4 Shift Instruction

32-bit instruction	16-bit instruction	Special case
SRAI	SRAI45	
SRLI	SRLI45	
SLLI	SLLI333	

Table 2.5 Bit Filed Mask Instruction

32-bit instruction	16-bit instruction	Special case
ZEB	ZEB333	
ZEH	ZEH333	
SEB	SEB333	
SEH	SEH333	
ANDI	XLSB33	
ANDI	X11B33	

Table 2.6 Branch and Jump Instruction

32-bit instruction	16-bit instruction	Special case
BEQ	BEQS38	Branch on Equal Implied R5
BNE	BNES38	Branch on Not Equal Implied R5
BEQZ	BEQZ38	
BNEZ	BNEZ38	
J	J8	
JR	JR5	
JRAL	JRAL5	

Table 2.7 Load/Store Instruction

32-bit instruction	16-bit instruction	Special case
LWI	LWI450	
	LWI333	
	LWI37	Load Word with Implied FP
LWI.bi	LWI333.bi	
LHI	LHI333	
LBI	LBI333	
SWI	SWI450	
	SWI333	
	SWI37	Store Word with Implied FP
SWI.bi	SWI333.bi	
SHI	SHI333	
SBI	SBI333	

Table 2.8 Compare and Branch Instruction

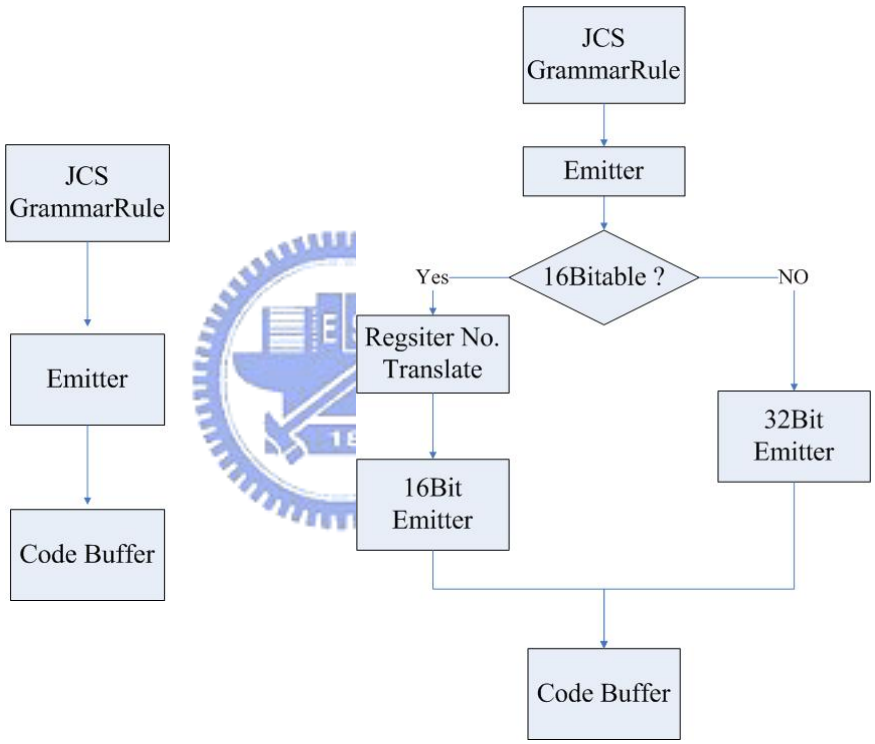
32-bit instruction	16-bit instruction	Special case
SLTI	SLTI45	
SLTSI	SLTSI45	
SLT	SLT45	
SLTS	SLTS45	
BEQZ	BEQZS8	Branch on Equal Zero Implied R15
BNEZ	BNEZS8	Branch on Not Equal Zero Implied R15

Chapter 3
The Multiple Fixed-width ISA Emitter

The Multiple Fixed-width ISA Emitter can emit 32-bit and 16-bit instructions in any desired mixture. The register manager will assign a register to a particular instruction and then the emitter will determine if a 16-bit instruction can be use. If not, a 32-bit instruction will generated instead. When a 16-bit instruction is to be generated, the register number must be converted according Table 2.1. Because, in Andes, the memory address accessed by a 32-bit memory instruction must be word-aligned, when the emitter wishes to write a 32-bit instruction to the code, it has to break that instruction into two 16-bit half-words and write the two half-words separately in order to avoid a Data-Alignment exception. It is essential for the register manager to choose an appropriate register if the emitter attempts to generate 16-bit instructions. The JIT writer can set up four register sets (CVMCPU_PHI_REG_SET, CVMCPU_BUSY_SET, CVMCPU_NON_VOLATILE_SET, and CVMCPU_VOLATILE_SET) for the register manager to choose appropriate registers. We may tune the four register sets to emit as many 16-bit instructions as possible. For certain *patch points*, we must be sure that patch instruction has the same size with the original instruction.

3.1 Multiple Fixed-width ISA Emitter Introduction

While JCS rules select one instruction, the emitter will be called to emit the instruction to code buffer. The Multiple Fixed-width ISA Emitter adds a test ("16-bitable" in Figure 3.1(b)) to determine if the emitter can emit 16-bit instruction. If so, it will translate the 32-bit instruction to the corresponding 16-bit instruction.



(a) (b)

Figure 3.1 (a) Original emitter. (b) Adding the "16-bitable" test.

3.1.1 Determine Instruction

In Andes ISA, there are six formats for 16-bit instructions---333–form, 45-form, 37-form, 38-form, 8-form, and 55-form. (333-form and 45-form are the two most popular formats for 16-bit instructions.) Some 32-bit instructions even do not have the

16-bit counterparts. The emitter first needs to determine if a 16-bit instruction can be issued. Figures 3.2 (a) is the flow chart for testing the 333-form and Figure 3.3 (a) is the flow chart for testing the 45-form. For example, in Figure 3.2 (b), an add instruction has registers R0 and R1 and the immediate value imm. R0 and R1 fall in the ranger for registers in an addi333 instruction. Furthermore, if the immediate value is no more than 7 (0x111), this instruction will be translated into a 16-bit instruction in the 333-form.

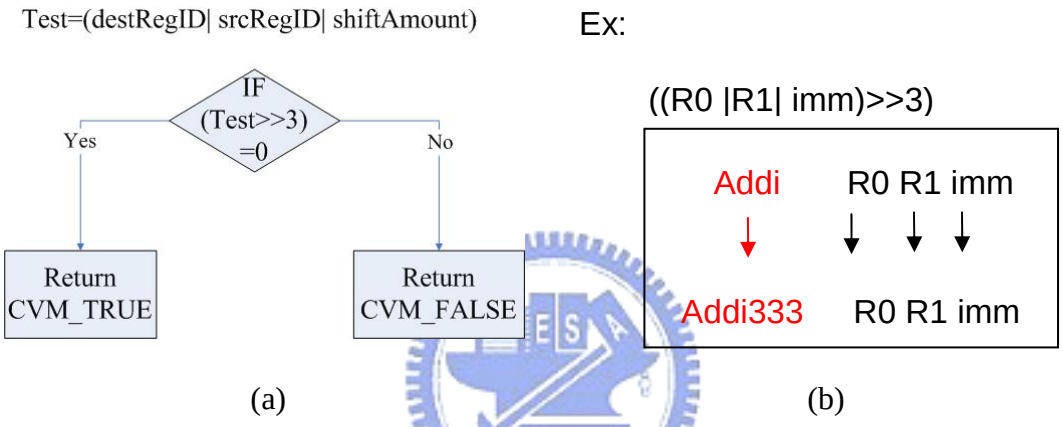


Figure 3.2 (a) Flow chart of testing the 333-form. (b) An example of Addi333.

When the immediate value is larger than 7, the emitter will try other forms, say the 45-form (4 bits for specifying a register and 5 bits for specifying the immediate value.) Figure 3.3 (a) shows the flow chart for testing if the 45-form can be used. There are other forms for 16-bit instructions. The emitter will try each form in turn. When no 16-bit form is applicable, a 32-bit instruction will be issued instead.

3.1.2 Translating Registers

A register may be encoded in 3, 4, or 5 bits according to the selected instruction formats. The encoding is shown in Table 3.1. For example, R17 is encoded as 10001 (T1) in 5 bits and as 1101 (H13) in 4 bits.

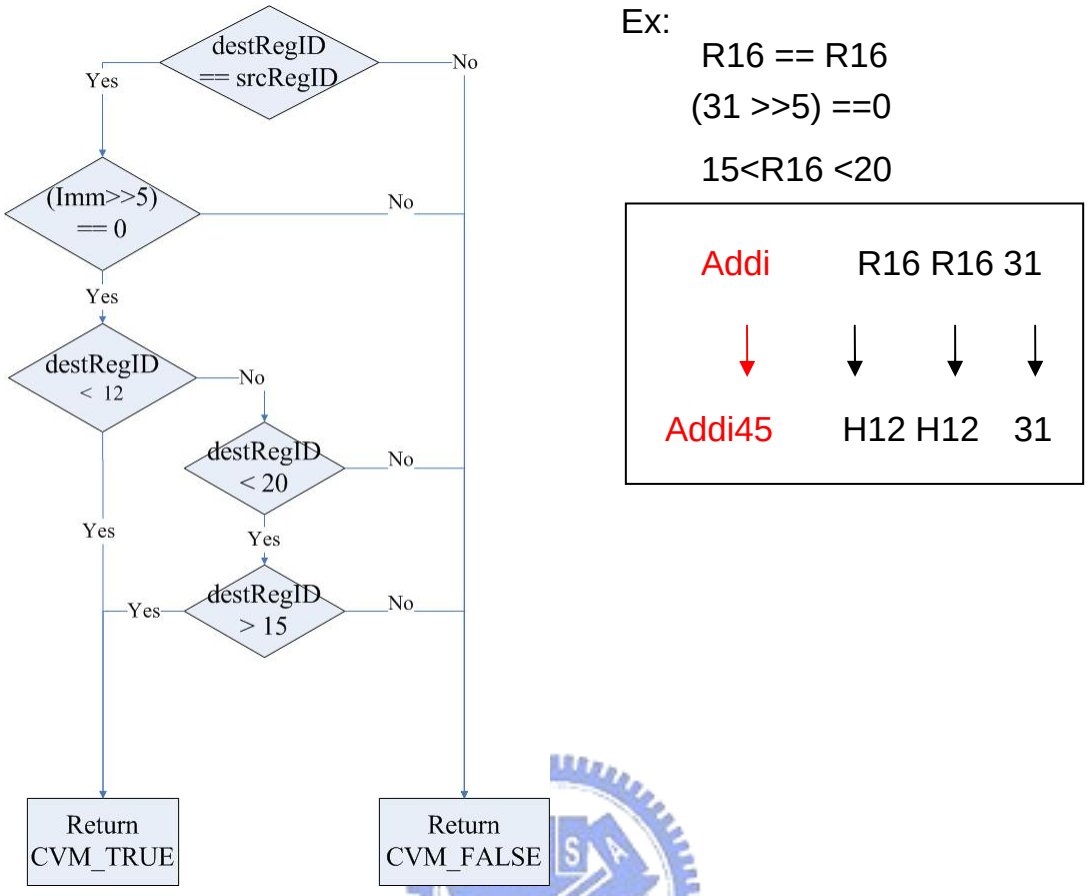
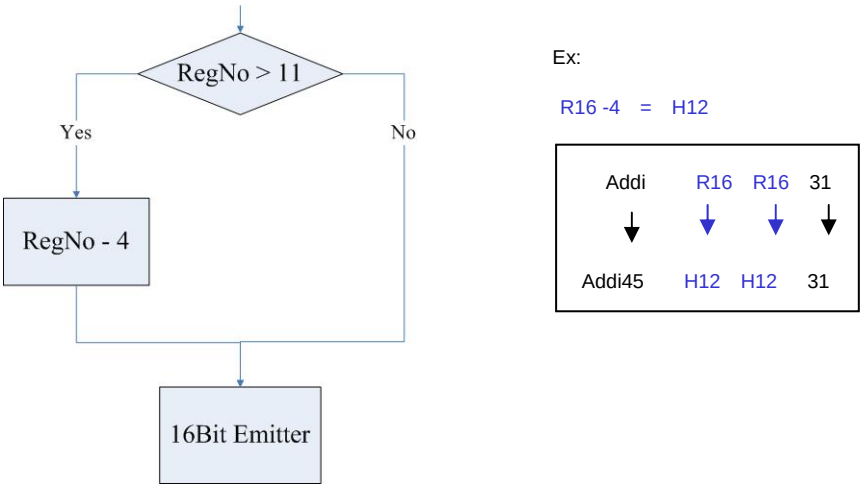


Figure 3.3 (a) Flow chart for testing the 45-form. (b) An example of ADDI45.

When the emitter wants to emit a 16-bit instruction, the emitter will test if the register assigned by the register manager could be used in a 16-bit instruction. For example, the 333-form is restricted to use registers R0 through R7 while the 45-form can use only registers R0-R11 and R16-R19 in the 4-bit field. (There is no restriction for the 5-bit field since 5 bits are enough to address any of the 32 general-purpose registers.) If the assigned register can fit in a 16-bit instruction form, then the emitter will translate the encoding of the register according to Table 3.1. This means that R16-R19 will be translated into H11-H15. The flowchart for the translation is shown in Figure 3.4 (a). The used registers of different mode are shown in Figure 3.5.

Table 3.1. The difference of two kinds of register set.

Register 32/16	32/16-bit (5 bits)	16-bit (4 bits)
R0	A0	H0
R1	A1	H1
R2	A2	H2
R3	A3	H3
R4	A4	H4
R5	A5	H5
R6	S0	H6
R7	S1	H7
R8	S2	H8
R9	S3	H9
R10	S4	H10
R11	S5	H11
R16	T0	H12
R17	T1	H13
R18	T2	H14
R19	T3	H15



(a) (b)

Figure 3.4 (a) Flow chart for translating register encoding. (b) An example of register translation.

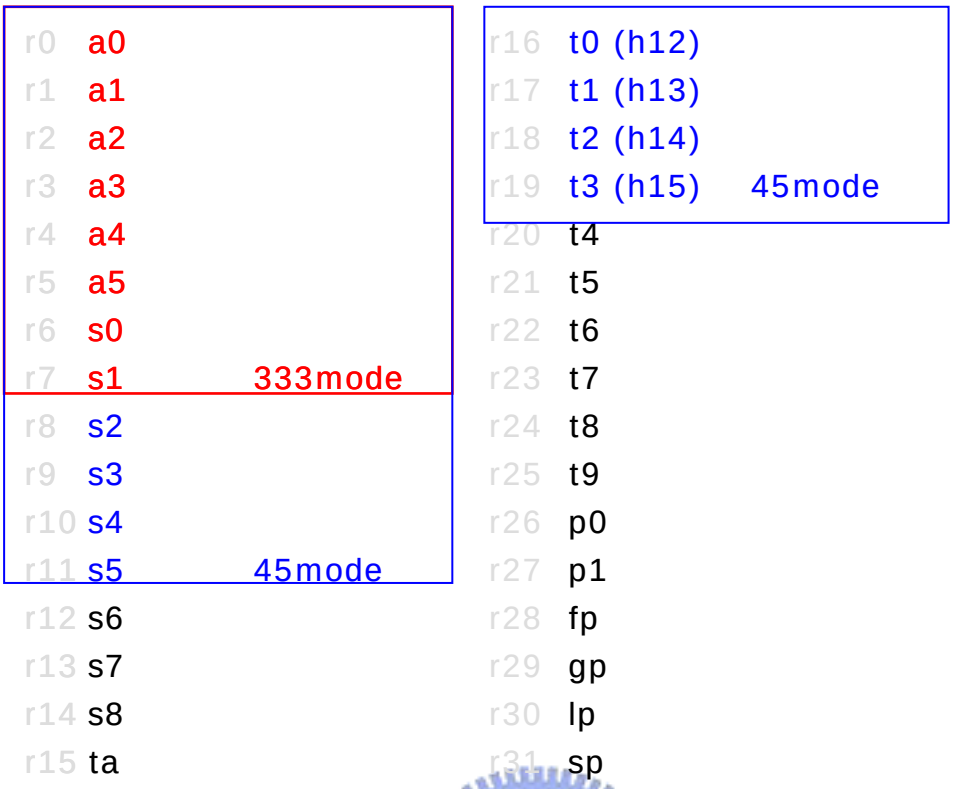


Figure 3.5 Register range of 333 mode and 45 mode

3.1.3 Instruction Alignment

In Andes, there is a restriction that the memory address accessed by a 32-bit memory instruction (Load/Store) must be word-aligned, that the least significant two bits of the address must be 0. When the emitter wants to place a 32-bit instruction into the code buffer, it will break the instruction into two half-words. Each half-word is written into the code buffer separately. This is explained in Figure 3.6.

3.2 Register Setting

A JIT writer may adjust the register setting to emit more 16-bit instructions. There are two places in the JIT that can be adjusted: the VM register set and the four code generator register sets.

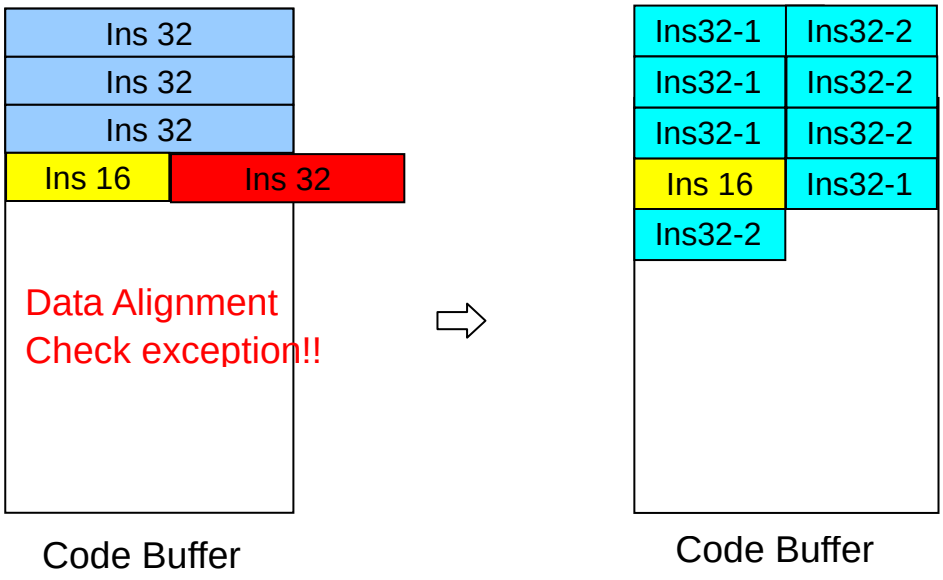


Figure 3.6 Avoid the Data Alignment Check exceptions when writing a 32-bit instruction into code buffer.

3.2.1 The VM Register Set

The VM register set contains four special registers: JSP_REG, JFP_REG, CHUNKEND_REG, and CVMCPU_EE_REG. They must be mapped to Andes registers properly. In our emitter, we use register FP for JFP_REG because it can use the special 37-form instructions.

Table 3.2. VM Register Setting

VM Register	Register
JSP_REG	R11
JFP_REG	FP
CHUNKEND_REG	S2
CVMCPU_EE_REG	S3

3.2.2 Code Generator Register Set

There are four *code generator register sets*: CVMCPU_PHI_REG_SET, CVMCPU_BUSY_SET, CVMCPU_NON_VOLATILE_SET, and CVMCPU_VOLATILE_SET in the header file jitrisc_cpu.h. The four register sets are used by the register manager to set up CVMRM_ANY_REG_SET, CVMRM_SAFE_SET, and CVMRM_UNSAFE_SET. (The CVMRM_EMPTY_SET is always an empty set.) When the JCS rules requests for a register, the register manager will select a register out of one of these four register sets. We wish to distribute the registers that can be used to generate 16-bit instructions into these four sets so that such a register is available when JCS rules requests for a register. The best distribution should be determined by extensive benchmarks. Currently, the distribution is shown in Table 3.3.

The register manager sets up the four sets CVMRM_ANY_REG_SET, CVMRM_SAFE_SET, CVMRM_UNSAFE_SET, and CVMRM_EMPTY_SET as follows. The CVMRM_EMPTY_SET is always an empty set. The CVMRM_ANY_REG_SET includes all registers except those in the CVMCPU_BUSY_SET. The CVMRM_SAFE_SET includes all the registers that are in both CVMCPU_NON_VOLATILE_SET and CVMRM_ANY_SET. Equivalently, the CVMRM_SAFE_SET includes all the registers that are in CVMCPU_NON_VOLATILE_SET but not in CVMCPU_BUSY_SET. The CVMRM_UNSAFE_SET includes all the registers that are in both CVMCPU_ VOLATILE_SET and CVMRM_ANY_SET. Equivalently, the CVMRM_UNSAFE_SET includes all the registers that are in CVMCPU_ VOLATILE_SET but not in CVMCPU_BUSY_SET. Table 3.4 summarizes the above specification in the register manager.

Table 3.3. RISC_CPU Register Setting

RISC_CPU Register Set	Register
CVMCPU_PHI_REG_SET	S1, S4, S5 ,S6 ,S7 ,S8 ,GP
CVMCPU_BUSY_SET	TA, P0, P1, FP
CVMCPU_NON_VOLATILE_SET	S0-S8, FP, GP
CVMCPU_VOLATILE_SET	ALL & ~CVMCPU_NON_VOLATILE_SET

Table 3.4. Register Manager register setting

JIT RegMan Register Set	Register set
CVMRM_BUSY_SET	CVMCPU_BUSY_SET | 1U<<CVMCPU_SP_REG | 1U<<CVMCPU_JSP_REG | 1U<<CVMCPU_JFP_REG | CVMRM_CHUNKEND_BUSY_BIT | CVMRM_CVMGLOBALS_BUSY_BIT | CVMRM_EE_BUSY_BIT | CVMRM_CP_BUSY_BIT | CVMRM_GC_BUSY_BIT
CVMRM_ANY_REG_SET	ALL &~(BUSY_SET)
CVMRM_SAFE_SET	(CVMCPU_NON_VOLATILE_SET & CVMRM_ANY_SET)
CVMRM_UNSAFE_SET	(CVMCPU_VOLATILE_SET & CVMRM_ANY_SET)
CVMRM_EMPTY_SET	Always empty set

3.3 Instruction Patch and Adjust

While the emitter emits a forward branch or jump to glue code, the address field in this instruction will be patched later. Since we do not know the size of the actual offset in the instruction, to be on the safe side, we always use 32-bit instructions for forward branch or jump to glue code.

Furthermore, the instructions for null check may also need additional patches. It is discussed in Sections 3.3.3.

3.3.1 Forward Branch

When the emitter emits a branch instruction with unknown offset, it will always issue a 32-bit instruction. The address field in this instruction will be patched later when the address of the branch target is known. Figure 3.7 shows that patch a forward branch instruction.

3.3.2 Glue Code

Sometimes the program has to calculate certain special values when it reaches a particular instruction the first time. (Ex. ResolveMethodTableOffsetGlue) The emitter will issue a “Jarl .glue” instruction to force the program to jump to the glue code. The special value is calculated in the glue code. At the end of the glue code, the calculated vale will be written to the word immediately following the “Jarl” instruction and the “Jarl” instruction is changed to a “J .skip” instruction. Having done that, the program continues execution following the “Jarl” instruction. Note that the glue code is executed only the once during program execution because it is a waste of time to calculate the same special value more than once. Changing the “Jarl” instruction to “J .skip” instruction can prevent the glue code being executed

again. Figure 3.8 shows the execution of glue code. Note that the “Jarl” instruction is changed to a “J .skip” instruction after the glue code is executed. A variation of glue code does not compute a special value; however, it is also executed only once—the first time it is encountered. This variation of glue code also needs patching as described above.

Due to the existing implementation of glue code (which was written in the assembly language for the 32-bit platform and always patched instructions at word-alignment), whenever a “Jarl” instruction may be patched by glue code, that “Jarl” instruction must be word-aligned. In this case, a two-byte “nop16” instruction might be inserted before the “Jarl” instruction in order to satisfy the requirement of word-alignment. This is because, in the existing glue code, instructions are always assumed to be word-aligned while in our target platform (Andes) instructions may be half-word aligned. In the future, we plan to rewrite glue code. Then the two-byte “nop16” instructions will become unnecessary. On the other hand, if the “Jarl” instruction will not be patched by the glue code, we can choose either a 16-bit (for half-word aligned) or a 32-bit (for word aligned) “Jarl” instruction. Note that the four reserved bytes (i.e., “.word ____”) following the “Jarl” instruction must always be word-aligned. The flow chart is shown in Figure 3.9. The list is shown in Table 3.5.

3.3.3 Trap-based Null Checks

Every time VM references an new object, the object must be check is null or not. While JIT wants to do null checks, a null-pointer trap will occurs, the return address (which is the address of the instruction immediately following the trapping instruction) will be saved in the link-pointer register (LP). If the trapping instruction is a 16-bit instruction, the return address is 2 plus the address of the trapping instruction. On the

other hand, if the trapping instruction is a 32-bit instruction, the return address is 4 plus the address of the trapping instruction. In Andes, an instruction is 16-bit if and only if the first (leftmost) bit of the instruction is 1. The flow chart is shown in Figure 3.10.

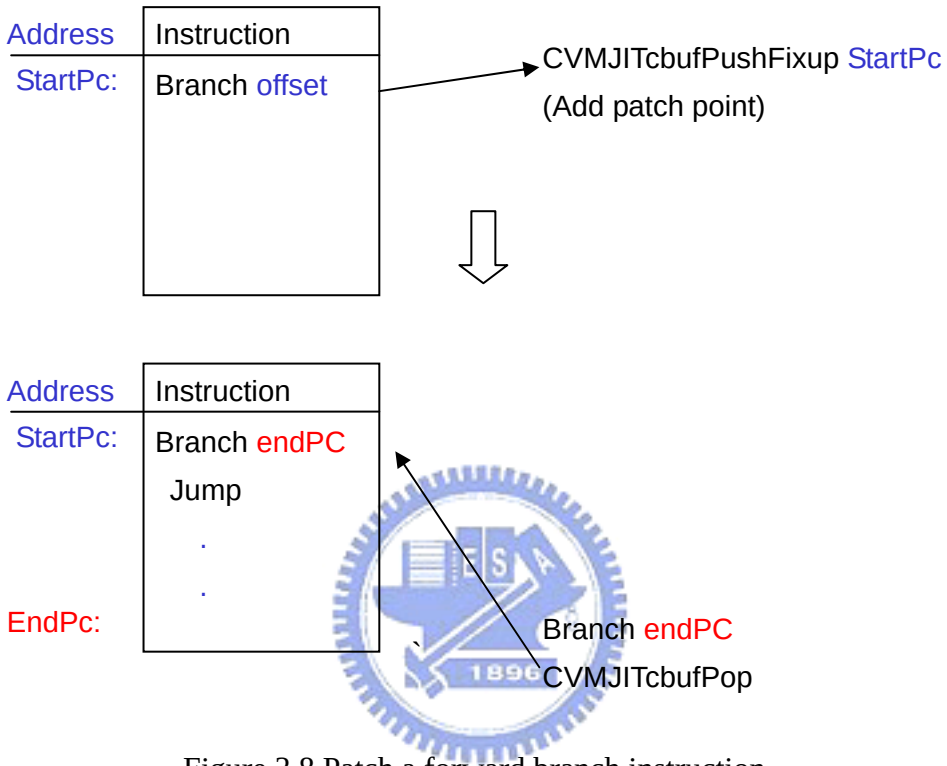


Figure 3.8 Patch a forward branch instruction

3.4 Summary

Our emitter will issue mixed 16-bit and 32-bit instructions in an attempt to reduce the resulting code size. Due to the alignment requirement in the existing JIT implementation, the emitter has to take care of the alignment of the issued instructions, adding “Nop” instructions when necessary. Because only some, but not all, registers can be used in 16-bit instructions, register allocations must be done carefully in order to generate more 16-bit instructions. We propose a simple heuristic for instruction translation in this thesis. In the next chapter, we will use benchmarks to verify the

usefulness of our heuristic for instruction translation.

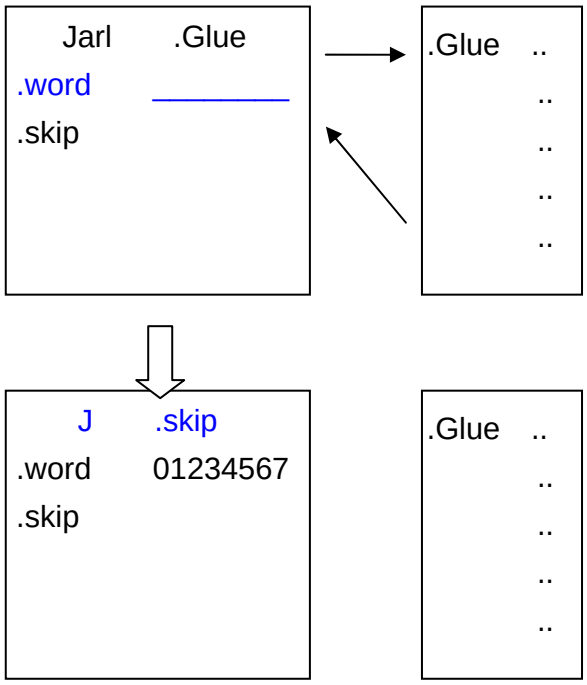


Figure 3.9 The execution of glue code. Yet another kind of glue code does not need patching the “Jarl” instructions. It functions like a subroutine.

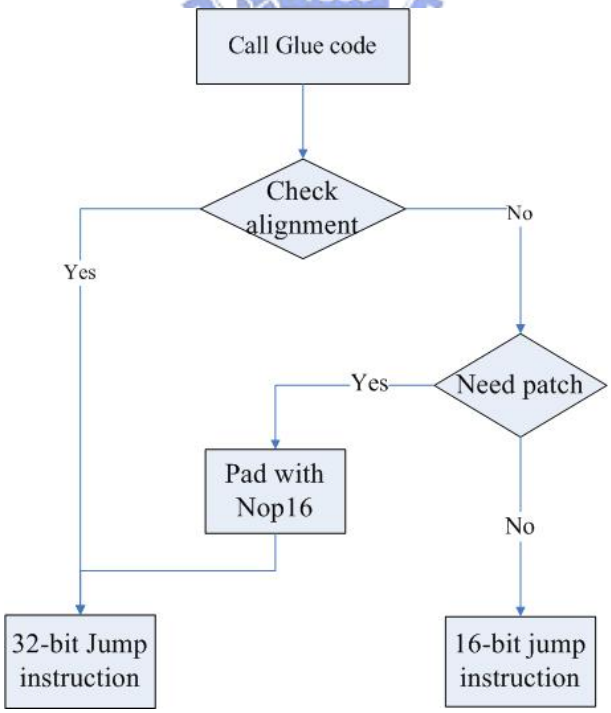


Figure 3.10 Adjust glue code flow chart

Table 3.5 Glue code list.

Case 1 : Only need .word after the call to be aligned.

Case 2 : Only jal/jral need be patch at runtime.

Case 3 : Not only .word after the call need to be aligned, but also jal/jral need to be patched at runtime..

Case 4 : Not only call instruction and .word after the call need to be aligned, but also the length of two instructions after the call need to be known at compilation time.

Case 1	Only need word-alignment
	CVMCCMruntimeLookupInterfaceMBGlue
	CVMCCMruntimeCheckCastGlue
	CVMCCMruntimeInstanceOfGlue
Case 2	Only be patch
	CVMCCMruntimeRunClassInitializerGlue
Case 3	Need Word-alignment and patched
	CVMCCMruntimeResolveNewClassBlockAndClinitGlue
	CVMCCMruntimeResolveGetstaticFieldBlockAndClinitGlue
	CVMCCMruntimeResolvePutstaticFieldBlockAndClinitGlue
	CVMCCMruntimeResolveStaticMethodBlockAndClinitGlue
	CVMCCMruntimeResolveClassBlockGlue
	CVMCCMruntimeResolveArrayClassBlockGlue
	CVMCCMruntimeResolveGetfieldFieldOffsetGlue
	CVMCCMruntimeResolvePutfieldFieldOffsetGlue
	CVMCCMruntimeResolveSpecialMethodBlockGlue
	CVMCCMruntimeResolveMethodBlockGlue
Case 4	Special Case
	CVMCCMruntimeResolveMethodTableOffsetGlue

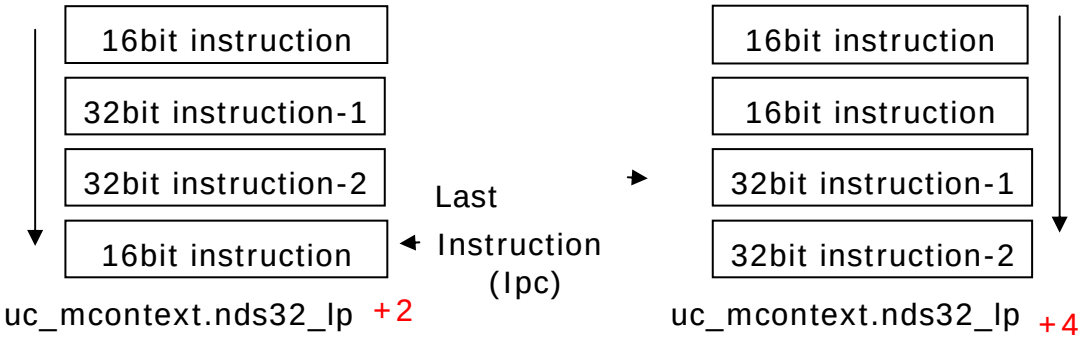


Figure 3.11 Determine the return address of a trap-based null check.

Chapter 4
Experiments Results and Analyses

4.1 Experimental Framework

In this section, we will show some experimental results of the Java JIT Compiler for 32bit-16bit Mixed Instruction Set Architectures. Experiments for this study were performed at the Andes ADP-AG101 platfrom at 400Mhz. (Figure 4.1) First, we design a lot experiment program for 16bit-32bit emitter to verify correctness. They test single target function of emitter Ex. Add operation. Next , we run a global test case: Testclass. Testclass provide by Sun Microsystems. Which Correctness are verified we run a lot of benchmark to collect date like code size and score (performance). The benchmarks are Embedded CaffineMark 3.0[19], CLDC Evaluation Kit and Grinder Bench[20]. Their program are shown in table 4.1, 4.2 and 4.3.

Table 4.1 Embedded CaffineMark 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 measure 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.

Table 4.2 CLDC Evaluation Kit

Name	Brief Description
Richards	Richards is a benchmark that simulates the task dispatcher in the kernel of an operating system.
DeltaBlue	DeltaBlue solves one-way constraint systems.
Queens	A solver of the n-queens problem. It is a classical problem used to illustrate several techniques such as general search and backtracking.
Image Processing	The Image Processing benchmark reads an image file and performs various transformations on it, such as Sobel, threshold, 3x3 convolver, and so forth.

Table 4.3 Grinder Bench

Name	Brief Description
Chess	A complete chess playing engine that is used to determine a set of chess moves.
Crypto	This suite of algorithms measures the performance of Java implementations in cryptographic transactions.
kXML	Measures XML parsing and/or DOM tree manipulation.
PNG	Shows how fast a Java implementation can decodes a PNG photo image of a typical size used on a mobile phone.

Figure 4.1 Andes AG101 Main Board

4.2 Correctness

Correctness is verified by SUN test class. It test 411 tests includes Null check, GC check, float number and others.

4.3 Compile Time

Figure 4.2 is the compiled time of two version of JIT. We can observe that the compile time between 32-only and mixed-ISA only has few increase. It mean low overhead of compile the code.

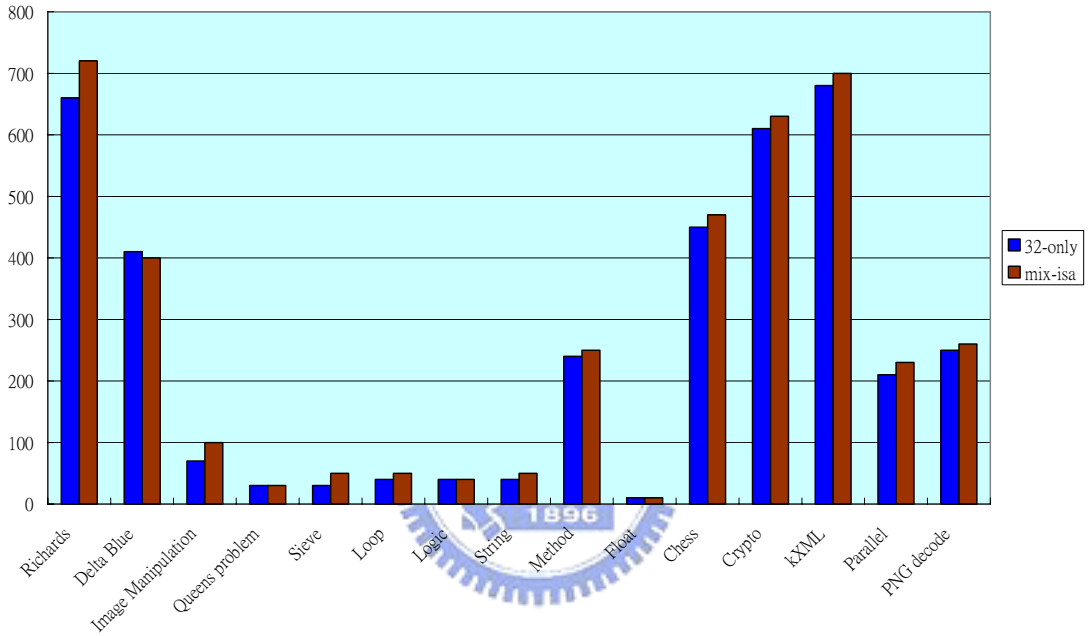


Figure 4.2 The compile time of benchmarks

4.4 Code Size

The reduce code size of the benchmark is shown in Figure 4.3. Average, we reduce almost 10% code size.

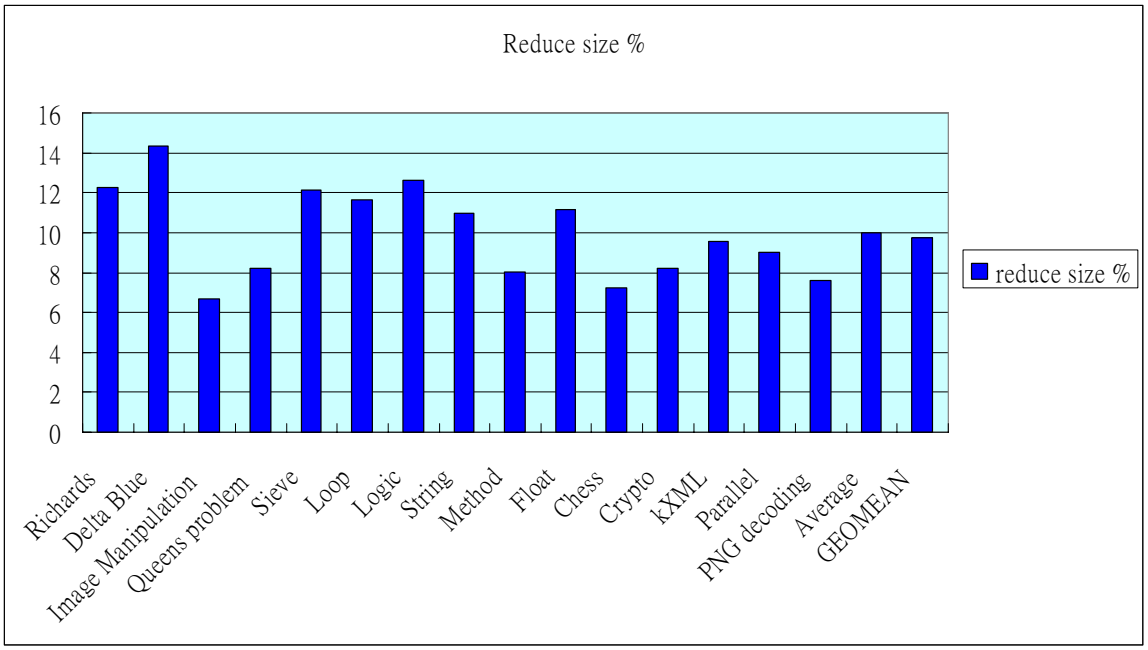


Figure 4.3 Code size of benchmarks

4.5 Performance

The Performance is shown in Figure 4.4. Our method decreases average 0.14% the performance of benchmarks. Specially, KXML program gets many performance's benefit. We analysis this program. Then we observe that KXML is the biggest program of our benchmark and it run longer time than others small programs. For other benchmark programs, the performance is actually *decreased*. The reason is that these benchmark programs run only for a very short time. The additional time we spent on run-time compilation dominates the overall performance. For a long running program, reducing the code size should result in significant performance improvement.

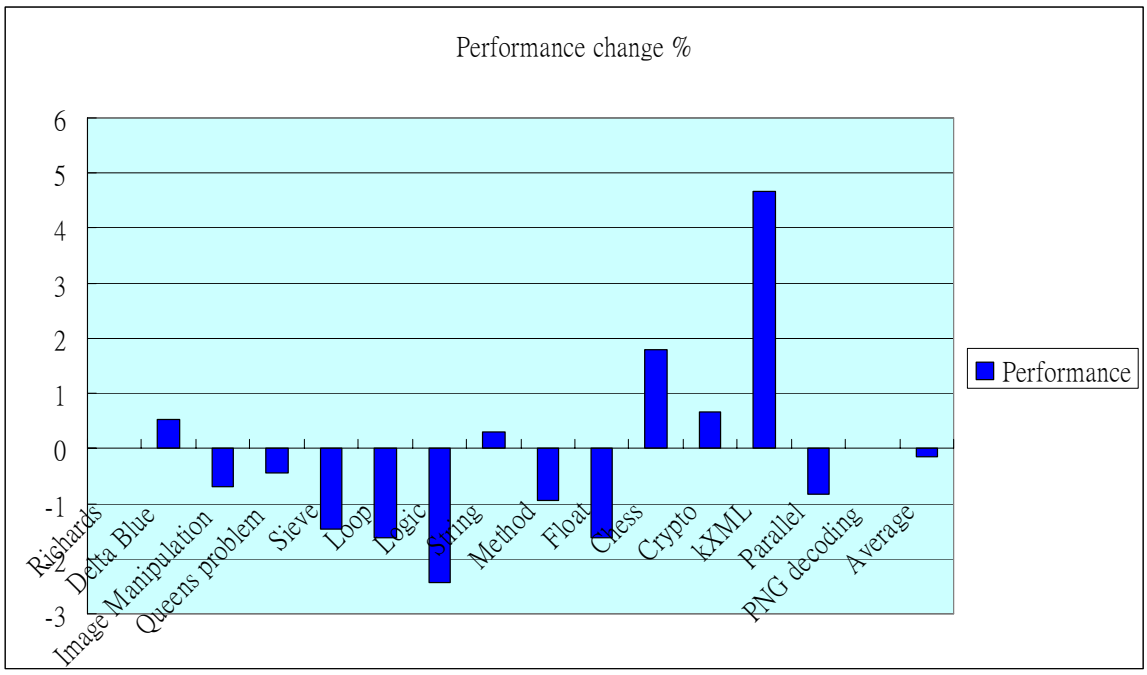


Figure 4.4 The performance of benchmarks

4.6 Summary

Our Java JIT compiler for 32bit/16bit Mixed Instruction Set Architecture successfully reduces code size by 10% on the average, with only slight additional compilation overhead. For a long running program, reducing the code size should result in significant performance improvement.

Chapter 5
Conclusion and Future work

This paper proposes an effectively method for reducing code size. This method is implemented in a Java JIT compiler. Our JIT compiler generates smaller code by making use of 32bit-16bit mixed instruction set than the original JIT that only uses 32-bit instruction set. Specially, the performance of the code generated by our method is almost equal to that by the original JIT compiler, sometimes even better. For a long running program, we expect the benefit will exceed the overhead.

There are a few slots in the code buffer that are reserved for the glue code to fill in appropriate offsets or modify the instructions at run time. The slots must be word-aligned in order to fit the existing glue code. Therefore, the emitter sometimes needs to add the nop instructions in the code buffer before the word-aligned slots. In the future, we can modify the glue code in order to remove the useless nop instructions.

In the process of emitting instructions, the registers assigned by the register manager decide if a 16-bit instruction can be used. In the future, the register-allocation algorithm deserves further investigation for performance improvement.

References

[1]. Sun Microsystems. Java ME CDC, http://java.sun.com/javame/technology/cdc, 2008

[2]. Sun Microsystems. Java ME, http://java.sun.com/javame , 2008

[3]. Sun Microsystems. CDC HotSpot Implementation Dynamic Compiler Architecture Guide, 2005.

[4]. Sun Microsystems. CDC Porting Guide, 2005.

[5]. Sun Microsystems. The CDC application management system, 2005.

[6]. Furber, S. 1996. ARM System Architecture. Addison-Wesley. ISBN 0-201-40352-8.

[7]. Goudge, L. and Segars, S. 1996. Thumb: Reducing the cost of 32-bit RISC performance in portable and consumer applications. In Proceedings of COMPCON.

[8]. Kissel, K. 1997. MIPS16: High-density MIPS for the embedded market. Tech. rep., Silicon Graphics MIPS Group.

[9]. Andes Technology. Andes Instruction Set Architecture Specification, 2007.

[10]. Andes Technology. Andes Programming Guide, June, 2007.

[11]. Lee, S., Lee, J., Park, C. Y., and Min, S. L. 2004. A flexible tradeoff between code size and WCET using a dual instruction set processor. In Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES). Amsterdam. 244–258.

[12]. Shin, I., Lee, I., And Min, S. L. 2002. Embedded system design framework for minimizing code size and guaranteeing real-time requirements. In Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS). Austin, TX. 201–211.

[13]. Lee, S., Lee, J., Min, S. L., Hiser, J., and Avidson, J. W. 2003. Code generation for a dual instruction set processor based on selective code transformation. In Proceedings of the 7th International Workshop on Software and Compilers for embedded Systems (SCOPES). Vienna. 33–48.

[14]. Naswamt, A. and Gupta, R. 2003b. Mixed width instruction sets. Communications of the ACM 46, 8 (Aug.), 47–52.

[15]. Krishnaswamy, A. and Gupta, R. 2003a. Enhancing the performance of 16-bit code using augmenting instructions. In Proceedings of the ACM SIGPLAN Conferece on Languages, Compilers,and Tools for Embedded Systems (LCTES). San Diego, CA. 254–264.

[16]. Halambi, A., Shrivastava, A., Biswas, P., Dutt, N., and Nicolau, A. 2002. An

efficient compiler technique for code size reduction using reduced bit-width ISAs. In Proceedings of the Design, Automation and Test in Europe (DATE). Paris.
[17]. Kirner, R. 2003. Extending optimising compilation to support worst-case execution time analysis.Ph.D. thesis, Vienna University of Technology.
[18]. Sheayun L, Jaejin L, Chang Yun Park, Sang Lyul Min, Selective Code Transformation for Dual Instruction Set Processors in ACM Transactions on Embedded Computing Systems, 2007
[19]. Pendragon Software Corporation, Embedded CaffeineMark 3.0 benchmark, http://www.webfayre.com, 1997
[20]. EEMBC. GrinderBench, http://www.grinderbench.co