A New Approach for Improving Ported Java JIT Compilers for Embedded Systems

Shuai-Wei Huang, Yu-Sheng Chen, Jean Jyh-Jiun Shann, Wei-Chung Hsu, Wuu Yang
Dept. of Computer Science,
National Chiao Tung University,
Hsinchu City 300, Taiwan
{swhuang, yusheng, jjshann, hsu, wuuyang}@cs.nctu.edu.tw

Abstract*-When a Java JIT compiler is ported to a new hardware platform, it usually cannot take full advantage of the special features of the new platform unless it undergoes thorough and massive optimizing. We propose a new approach to improve the code generator in a ported Java JIT compiler. A static code analyzer is used to automatically discover frequently-occurring patterns in the generated code that are suitable for peephole optimizations. Then the patterns are incorporated in the JIT compiler by modifying instruction selection rules and code emitters. The approach of automatically discovering patterns is feasible because (1) there does exist patterns in the code generated by most compilers and (2) a peephole optimizer requires only quite simple patterns, which can be discovered easily. Our target platform is the Andes architecture, which features several novel hardware facilities. The result of our experiment shows the approach is quite promising.*

Keywords: JIT compiler, peephole optimization, pattern matching, embedded systems, peephole optimizer.

1. Introduction

Many existing compilers for embedded systems generate low-quality code since the compilers, which are usually ported from different platforms, cannot take full advantage of the special features of the new platforms [8]. There is a need for further optimizations for the code generated by the compiler.

In this paper, we describe a new approach for optimizing ported compilers. We ported the CDCHI Java virtual machine [1][2][3][4] (which includes a Java JIT compiler) to the Andes platform [5][6]. For improving the code generator in a Java JIT compiler, we propose a new method. We implemented a *local code analyzer* and a *pattern-based peephole optimizer* that can automatically analyze the code generated by the ported JIT compiler for the Andes platform and help identify patterns of instruction that can be reduced to more efficient ones. The patterns are then implemented as JCS (Java Code Select, a code generator generator) rules or are incorporated into the code emitter in the ported Java JIT compiler.

Because of the similarity of Andes and MIPS ISA, we start porting with the MIPS version of CVM. After finish porting work, we observed that the quality of the code generated by the ported JIT compiler can be improved. It did not make use of the special features provided by Andes ISA. On the other hand, since the Andes platform is still in the development stage, Andes people are eager for our feedback concerning the Andes ISA. These reasons motivate us to develop a tool to analyze code generated by the ported JIT compiler and identify patterns in the generated code that can be optimized.

The rest of this paper is organized as follows. In Section 2, we briefly introduce the CDCHI virtual machine and the Andes architecture and review related work for peephole optimization. We describe, in Section 3, the optimization framework, the implementation details of the local code analyzer, and the pattern-based peephole optimizer. In Section 4, we discuss the effective patterns in the generated code and the results for the reduction of code sizes. Finally, we conclude the work in Section 5.

2. Related work

2.1 CVM Overview

The Connected Device Configuration HotSpot Implementation virtual machine (CDCHI VM, a.k.a. CVM) is designed for resource-constrained devices, such as consumer products and embedded

The work reported in this paper is partially supported by Andes Technology Corporation, and National Science Council(NSC), Taiwan, Republic of China, under grants NSC 95-2221-E-009-020 and NSC 97-2218-E-009 -029.

devices, including smart phones, personal digital assistants (PDA) and global positioning systems (GPS)[2].

A Java program is compiled to bytecode by a Java compiler. Bytecode is then executed by the CVM. When the bytecode of a method is executed more often than a pre-set threshold, the JIT compiler in the CVM will translate the bytecode into native code for the underlying hardware platform, which, in our case, is the Andes binary code.

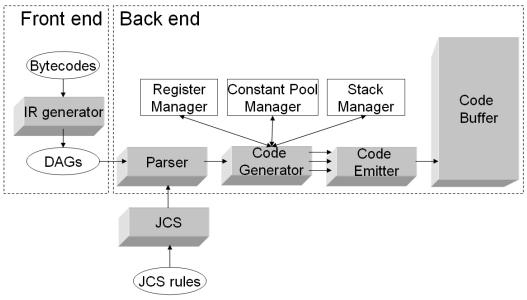

Figure 1. The JIT compiler in CVM

The JIT compiler consists of two parts (see Figure 1). The front end translates bytecode into intermediate representations (IR) and handles other issues, such as code verification and security checks, and numerous optimizations on the IR. Then, the back end parses IR and generates native code.

The parser for the IR is generated from many instruction selection rules by the JCS tool. Most JCS rules have semantic actions, and this is where the code generator takes place [3]. The code generator manages the registers in the Andes processor, the constant pool, and the run-time stack. Then, the code generator will call the code emitter to emit instructions. The code emitter is the last stage of the JIT compiler and generates instructions for the underlying processor.

2.2 Andes Architecture

The Andes ISA is designed in the RISC style and is equipped with 32 registers [5]. It provides various types of load/store instructions. The ".bi" form of a load or store instruction means "before increment", in which the base register will be updated after the memory operation. Instructions of the "before increment" type seldom occur in the code generated by the ported JIT because the original JIT (which was designed for MIPS) never generates such instructions since the MIPS processors do not provide instructions of this type.

Another seldom used instruction type is the conditional branch instructions. The ported JIT compiler generates only the "equal" or "not equal" forms of the conditional branch instructions. Since Andes ISA supports more types of conditional branch instructions than MIPS, there should be good opportunity to improve the quality of the generated code.

Many processors support special-purpose instructions, which are equivalent to a sequence of more primitive instructions. Special-purpose instructions are often smaller and/or faster than the more generic ones. In the Andes platform, the ported JIT compiler never emits the "load/store multiple word" instructions, "branch with link" instructions and "conditional move" instructions. Instead, the JIT compiler usually emits two or more instructions to achieve the effects of these special-purpose instructions. This means that we can find many such patterns in the code generated by the ported JIT compiler and transform them into the special-purpose instructions.

2.3 Peephole optimization

Peephole optimization has been studied since 1965 [9]. The success of a peephole optimizer depends on the time and space for recognizing redundant sequences of instructions. Davison and Fraser [10][11][12] introduced a machine-independent and retargetable peephole optimizer, which replaces adjacent instructions with an equivalent single instruction.

Peephole optimization introduced by Kessler [13] was, instead of hand-written, automatically generated from an architectural description and allowed optimizations across basic blocks. Using patterns matching for code optimization is still one of the most popular approaches [8][14]. Spinellis used string-pattern matching to find out patterns. A pattern is a regular expression. Recently, Kumar defined numerous finite automata to recognize patterns [8]. The finite automaton is good for recognizing patterns that are not adjacent. Kumar also provided a replacement algorithm for resolving optimization conflicts. In our research, we find patterns and resolve optimization conflicts in a recursive way.

3. Code Analyzer and Peephole Optimizer

We propose two techniques to help find redundant instructions and identify patterns that can be optimized. Based on optimizations on local code or adjacent instructions, we divide optimizations into two parts. First, the *local code analyzer* aims at discovering patterns and evaluating the benefits of the patterns in a

peephole optimizer. It implements several common compiler optimizations within basic blocks. In addition, the local code analyzer will determine sequences of contiguous instructions as base patterns. The *pattern-based peephole optimizer* repeatedly finds occurrences of patterns in the benchmarks. It reports the proportion and the number of occurrences of various patterns in the benchmarks. Useful patterns are selected based on the occurrence frequencies. We will modify the JCS rules and revise the emitter functions so as to generate efficient code sequences for useful patterns. The overall optimizer framework is shown in Figure 2.

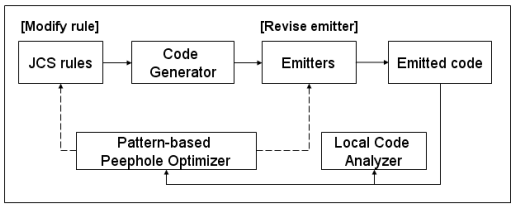

Figure 2. Overview of the proposed optimizing framework for CVM

3.1 Local Code Analyzer

Before discussing the local code analyzer, we define two terms first. If the value of a register will be updated after executing an instruction, we called the register a "*producer register*". On the other hand, if the value of a register is used but not updated, we called the register a "*consumer register*".

The optimization techniques which are implemented in the local code analyzer are classified into three categories: eliminating redundant instructions (which will reduce the code size), replacing with more efficient instructions (which will reduce the execution time), and supporting optimizations (which may increase the opportunities to eliminating redundant code).

Dead code elimination is a common compiler optimization. It is used to reduce code size by removing instructions that do not affect the programs [7]. For example, the instructions that define values in a producer register can be safely removed, if it is redefined before any using..

Redundant load/store elimination attempts to identify useless load/store instructions. We record the target register, base register, and the offset value in an instruction. If another instruction carries the same target register, base register, and offset value, it could be considered redundant.

Load copy optimization is implemented and applied in conjunction with redundant load/store elimination. This optimization will rewrite a load instruction as a move instruction if the value of a memory location has been loaded into another register.

Constant propagation, *copy propagation* and *common sub-expression replacement* can reduce the usage of consumer registers and improve the opportunities of redundant code elimination. If the instructions are adjacent, the propagation could also reduce data dependency.

These optimizations can help us to automatically find out reducible and optimizable sequences of successive instructions. Collected sequences (or patterns) are then feed into the pattern-based peephole optimizer to do further processing.

3.2 Pattern-base Peephole Optimizer

The pattern-based peephole optimizer, which is off-line, can be divided into four parts (see Figure 3): basic patterns, pattern-matcher generator, pattern matcher, and peephole optimizer. The *basic patterns* are collected from the local code analyzer. A sequence of instructions may match several patterns. The *peephole optimizer* will try all possible combinations of pattern matchings and yield the one with the highest gains. The peephole optimizer will also create a new pattern so that the (on-line) code generator will use the pattern directly without trying all combinations of pattern matchings for similar instruction sequences when the patterns are incorporated into the JCS rules. The *pattern matcher* is generated automatically from the patterns by the *pattern-matcher generator*. Each pattern has a corresponding function in the pattern matcher, which will be called by the peephole optimizer.

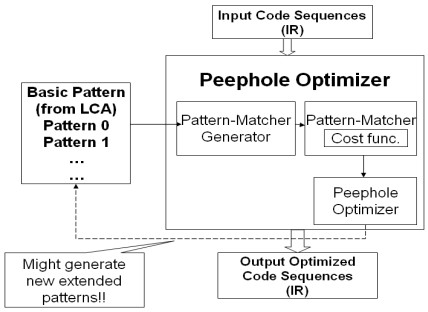

Figure 3. Framework of the pattern-based peephole optimizer

The pattern matcher has three tasks: *matching*, *replacing*, and *calculating gains*. Once a matching is found, the original instructions are replaced with more efficient ones. Then, we can calculate the

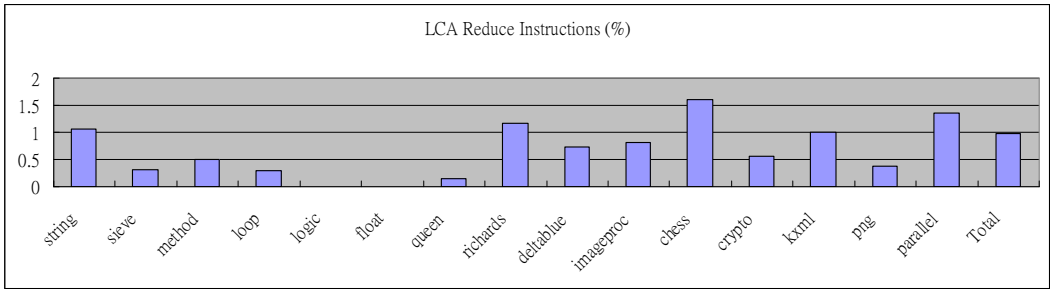

Figure 4. Total reductions by LCA

gains of replacement.

The input of the pattern-matcher generator is the descriptions of the instruction patterns. Each instruction of a pattern consists of an opcode and descriptors of operands. We support three types of descriptors: register, immediate, and address descriptors. A descriptor has its own ID. The "immediate" descriptor can be used to express not only any constant (e.g., c0, c1) but also the actual values (e.g., 0, 1).

In our implementation, all patterns are a sequence of contiguous instructions. When matching program code against patterns, the opcodes of the instructions must be identical and in the same order. Furthermore, the corresponding descriptors must also be matched. For instance, consider the fourth pattern in Table 3,

```
movi r0, 0
beqz r0, a0
```

This pattern will match

```
movi r5, 0
beqz r5, 0x1234
```

But it will not matching

```
movi r5, 0
beqz r6, 0x1234
```

There is a cost function in the pattern matcher. The optimizer calculates the gains of a pattern replacement based on the definitions of *reduction gain* and *replacement gain*. When multiple matchings are possible, the one with the highest gains is selected by the peephole optimizer.

After a matching is found, new instructions will replace original instructions. The optimizer will attempt to match the new instructions against the patterns. Repeated pattern matching essentially performs an exhaustive search for the highest gains.

4. Benchmarks and Results

For benchmarking, we selected fifteen programs from CLDC evaluation kit, Embedded Caffeine Mark [17] and Grinder Bench [16] (see Table 1). We ran all programs on Linux 2.6 on an Andes development board AG101. The clock rate of the on-board processor is 400 MHz.

The local code analyzer gathers statistics of the number of eliminated instructions and examines the patterns found in the benchmarks. Then these patterns are incorporated into our ported JIT compiler. Performance improvement is then measured.

Table 1. Benchmark programs

Benchmark	Programs
CLDC evaluation kit	Richards, DeltaBlue, ImageProc, Queen
Embedded Caffeine Mark	Sieve, Loop, Logic, String, Method, Float
Grinder Bench	Chess, Crypto, kXML, Parallel, PNG

4.1 Redundant Code Elimination

Since the order of optimizations may affect the number of eliminated instructions, we tried various combinations of five optimizations (dead code elimination, redundant load/store, copy propagation, constant propagation, and common sub-expression). Figure 4 shows the highest reduction ratios (with the order: Redundant load/store elimination, dead code elimination, copy propagation, and dead code elimination) for the benchmarks. The average reduction ratio (in terms of code size) is 0.98 %. The chess program has the highest reduction ratio: in total, 1.6% of instructions are eliminated.

4.2 Patterns

From our experiment, we discovered that some patterns occur very frequently in the emitted code. The frequent patterns can be replaced with more efficient instructions. The code generator in the JIT can be improved by incorporating these patterns into JCS rules, modifying the emitter accordingly and implementing delayed emission.

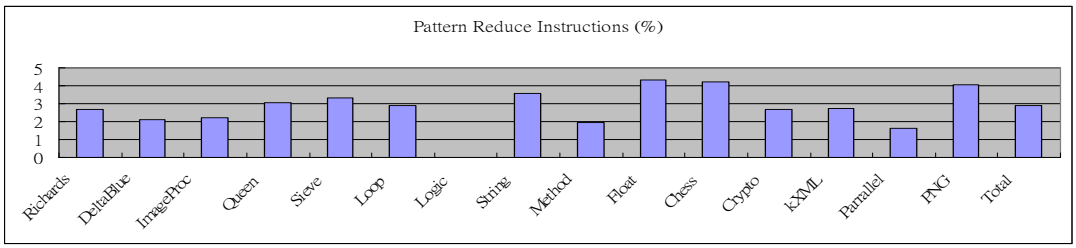

Figure 5. Total reduction by rewriting JCS rules and emitters

The most frequent pattern is "pre-decrement offset followed by a load instruction". This pattern occurs when a called method is returned and loads the return value from the stack frame. The JIT compiler emits an instruction for manual pre-decrement and a load instruction followed. The pair of instructions can be rewritten as a "lmw" instruction with load after decrement and suffix.

Another common pattern is the conditional branch instructions. In this pattern, the "slt/slts" instructions can be replaced with the "slti/sltsi" instructions. This pattern is emitted for the "BCOND_INT" IR node in code generator. For the example in Table 2, we can observe that the $ta register is assigned a constant 6 (the "movi" instruction) and is compared to the $s6 register (the "slt" instruction). For this case, we can use the "slti" instruction (set-if-less-than-an-immediate). Of course the "bnez" instruction should be replaced with the "beqz" instruction. The instruction count is reduced by 1.

Table 2 Patterns of conditional branch

Original pattern			
movi	$ta,	6	
slt	$ta,	$ta,	$s6
bnez	$ta,	0xf77aba74	
Replacement pattern			
slti	$ta,	$s6,	7
beqz	$ta,	0xf77aba74	

The third pattern is a special case of "setting a register with the address of a CVMObjectICell". A CVMObjectICell structure of CVM holds a pointer to an object [3]. In this pattern, the instructions first load the address of an object and then load the contents from that address. The address of the object will be loaded into a register through the "sethi" and "ori" instructions. Then, the loaded register serves as the base register in the following "lwi" instruction. Since the "ori" instruction is intended to set the lower half of the register, it can be combined with "lwi" instruction since the lengths of the immediate fields of "ori" and "lwi" are equal. For example, the following three instructions:

```
sethi   r5, 12345
ori     r5, r5, 678
lwi     r6, [r5+0]
```

can be replaced with the following two instructions:

```
sethi   r5, 12345
lwi     r6. [r5+678]
```

Table 3 shows some of the patterns in Andes native code generated by JIT compiler that are collected by the local code analyzer and the peephole optimizer. Note the number of eliminated instructions for each pattern.

Table 3. Patterns found from Andes assembly code and their respective gain

Original Patterns	Replacement Patterns	Reduction in the number of instructions
addi r0, r0, -4 lwi r1, [r0+0]	lmw.adm r1, [r0], r1 DELETE	1
lwi r1, [r0+0] addi r0, r0, c0	lwi.bi r1, [r0], c0 DELETE	1
movi r0, c0 slt r1, r0, r2 bnez r1, a0	DELETE slti r1, r2, c0 + 1 beqz r1, a0	1
movi r0, 0 beqz r0, a0	j a0 DELETE	1
movi r0, 0 bnez r0, a0	DELETE DELETE	2
lwi r0, r1, c0 lwi r0, r2, c1	DELETE lwi r0, r2, c1	1
swi r0, r1, c0 lwi r0, r1, c0	swi r0, r1, c0 DELETE	1
addi r0, r0, 0	DELETE	1

4.3 Rewriting JCS Rules and the Emitter

Most patterns can be incorporated into the JIT compiler by rewriting the JCS rules. Then the JCS tool is used to generate the new code generator. Generally, we also need rewrite the emitter so that it can cooperate with the new code generator.

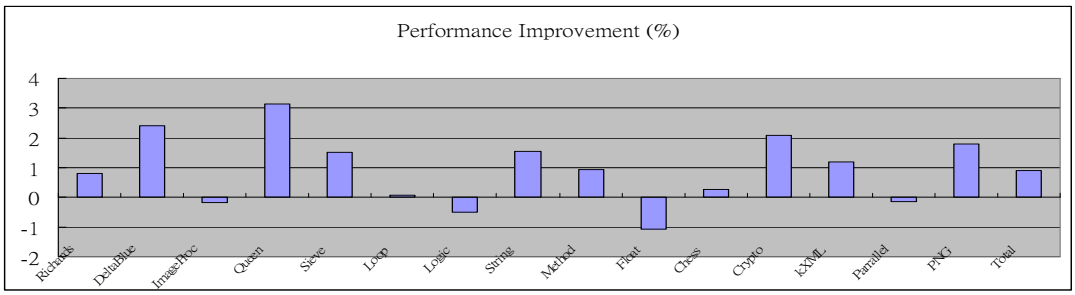

Figure 6. Performance improvement after rewriting JCS rules and emitters

Another way to improve the JIT compiler is the *delayed emission*, which is easy to implement and can work for all patterns, but it comes with a higher penalty. Most of the time, the more frequent patterns are generated within semantic actions or the emitter function.

In the revised JIT compiler, we add three of the most frequent patterns (load-multiple-word, conditional branch, common sub-expression). On average, 2.91% of the instructions can be eliminated (see Figure 5). The performance improvement (in terms of execution time) of all programs is 0.89% (see Figure 6) after rewriting the JCS rules and the emitter.

5. Conclusion and Future Work

Rewriting JCS rules and the emitters could improve our system performance if the optimized method is *hot* enough. The added overhead in the emitter and the frequency of the patterns are the key points for improving performance. If a method does not run for a long time, the overhead will lower the performance.

In the process of identifying patterns, we observe that some instructions are difficult to use. Providing new instructions or revising the original ones may help reduce code size and improve performance. For example, the "conditional branch and link" instruction is never emitted by the JIT compiler since the type of instruction only supports the greater-or-equal and less-then condition. But for all programs, we see that the "bnez" and "beqz" instructions are emited the most often and are frequently followed with a "jal" instruction. If we can support "beqzal" and "bnezal" instructions, we can reduce the code size even more (estimated at 2.52% reduction in code size). We will gather statistics of the frequencies of instruction pairs to evaluate the benefits of the new instructions in the future.

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

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

[7] S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc, August 1997.

[8] Rajeev Kumar, Amit Gupta, BS Pankaj, Mrinmoy Ghosh, and PP Chakrabarti, "Post-compilation optimization for multiple gains with pattern matching," *ACM SIGPLAN Notices* vol. 40, no. 12, pp.14 - 23, December, 2005.

[9] W. M. McKeeman. "Peephole optimization," Comm. *ACM* vol. 8, no. 7, pp.443-444, July, 1965.

[10] J. W. Davidson and C. W. Fraser. "Automatic generation of peephole optimizations," *In Proceedings of Best of PLDI*, pp.104-111, 1984.

[11] J. W. Davidson and C. W. Fraser. "Eliminating redundant object code," *In Ninth Annual ACM Symposium on Principles of Programming Languages*, pp.128-132, 1982.

[12] J. W. Davidson and C. W. Fraser. "Code selection through object code optimization," *ACM Trans. Programming Languages and Systems*, vol.6, no.4, pp.505 - 526, October, 1984.

[13] P. B. Kessler. "Discovering machine-specific code improvements," *In Proc. Symp. Compiler Construction. ACM SIGPLAN Notices*, vol.21, no.7, pp.249 - 254, July, 1986.

[14] Diomidis Spinellis. "Declarative peephole optimization using string pattern matching," *ACM SIGPLAN Notices*, vol.34, no.2, pp.47–51, February, 1999.

[15] Sorav, Bansal, and Alex Aiken. "Automatic Generation of Peephole Superoptimizers," *In Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, October, 2006

[16] EEMBC. GrinderBench, http://www.grinderbench.com

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