行政院國家科學委員會專題研究計畫　期中進度報告

記憶體受限之 Java Just-in-time (JIT)編譯器設計(1/3)

計畫類別： 個別型計畫
計畫編號： NSC93-2213-E-009-078-
執行期間： 93 年 08 月 01 日至 94 年 07 月 31 日
執行單位： 國立交通大學資訊工程學系(所)

計畫主持人： 單智君
共同主持人： 鍾崇斌
計畫參與人員： 陳裕生、劉彥志、黃欽毓、黃俊諭

報告類型： 精簡報告

處理方式： 本計畫可公開查詢

中　華　民　國 94 年 6 月 1 日

一、中英文摘要

中文摘要

小型電子裝置需求急速增加，為了要執行 Java 程式，必須在這些小型系統上執行 Java 虛擬機器。然而，一般以 Interpreter 的方式來直譯 Java Bytecode 會嚴重影響執行效能；因此，在兼具執行效能和程式可攜性的考慮下，可以選擇以 Just in time compiler (JITC) 為虛擬機器的執行引擎。我們將此種型態的 JITC 稱之為記憶體受限之 JITC (MCJITC)。

本計畫今年度針對符合 J2ME 規格的 KVM 為基本平台，在其上實作 MCJITC。我們研究與實作的重點主要在於 IR Generation、Code Generation、以及相關的最佳化技術。

我們在 Sun KVM 1.0.4 版本上實作完成了 Mix-Mode Memory-Constrain JIT Compiler (MCJITC)並且加入 instruction folding of stack operations 與 rule-based null point check elimination 等最佳化技術 。實驗結果顯示，我們的 MCJITC 比原始 KVM 快 2.09 倍。

Keywords：即時編譯器、Java 虛擬機器、KVM/CVM、ARM 處理器、J2ME

英文摘要

Many small electronic devices are increasing rapidly. In order to execute Java programs, Java Virtual Machine (JVM) needs to be executed in these miniature systems. However, common interpretation execution will degrade execution performance seriously. Therefore, under the consideration of both execution performance and portability, the execution engine of Virtual Machine is considered to be the Just-In-Time Compiler (JITC). We call this type of JITC as memory constrained JITC (MCJITC).

In this year’s project, we aim at implementing a working MCJITC based on the existing KVM from Sun Microsystem, which must also adhere to J2ME specification. Our research and implementation focus on IR generation, code generation and related optimization techniques.

We have implement the Mix-Mode Memory-Constrain JIT Compiler (MCJITC) based on version 1.0.4 of Sun's KVM, and employ some optimizations including instruction folding of stack operations and rule-based null point check elimination in our MCJITC. Simulation results show that our MCJITC is average 2.09 faster over the pure interpreter.

Keywords：JIT compiler、Java Virtual Machine、KVM/CVM、ARM processor、J2ME

二、報告內容

(一) Introduction

Many small electronic devices are increasing rapidly. In order to execute Java programs, Java Virtual Machine (JVM) needs to be executed in these miniature systems. However, common interpretation execution will degrade execution performance seriously. Therefore, under the consideration of both execution performance and portability, the execution engine of Virtual Machine is considered to be the Just-In-Time Compiler (JITC).

(二) Research Objectives

Our objective is to design and implement an embedded JVM (MCJITC), which is small footprint compared to other existing embedded JVMs. We employ mixed-mode execution in our embedded JVM and further facilitate the common optimizations, aiming at striking a balance between speed performance and memory usage.

(三) Related Researches Discussion

Mixed-mode JITC Architectures

In order to setup our MCJITC environment, we use mixed mode bytecode execution [1], which mixed interpreter and JITC compiled code execution of java bytecodes, as our basic execution engine.

JITC Optimizations [2-4]

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.

(四) Research Methods

4.1 MCJITC Architecture

In this section, we detail the design of the IR generator and the native code generator in the MCJITC. In addition, in order to reduce compilation cost and to keep the MCJITC small-footprint, several design decisions are made based. 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 4-1 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 reason- able for portability. The IR generator is responsible for translating Java bytecode into machine-independent three-address IR, and

therefore is portable across platforms.

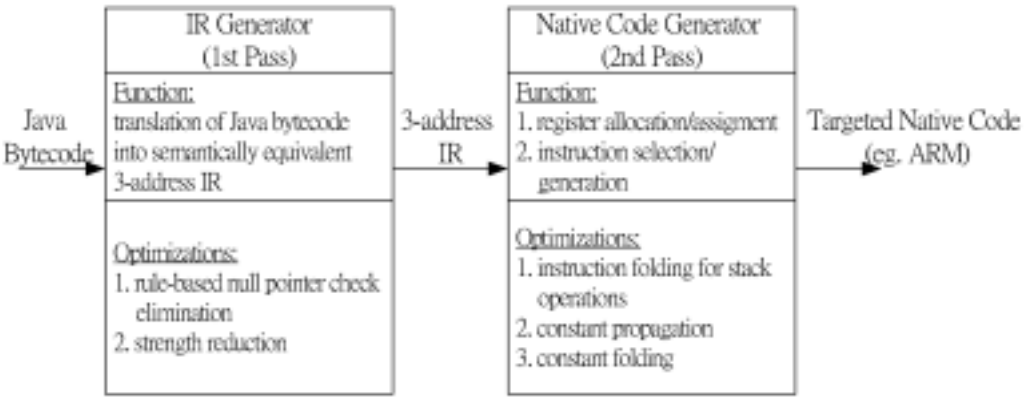

Figure 4-1 Two-pass Compiler Architecture

• 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.

• 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 de-virtualization, synchronization, object construction/destruction, and etc. As a result, these complex bytecode instructions are considered to be non-compile-able in the MCJITC.

4.1.1 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 their relative addresses are also fixed. 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.

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. 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

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.
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 program has branch-type instructions, its control flow is not always sequential. In order to overcome this problem, it is necessary 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 [5]. 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.

4.1.2 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 environment.

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 MCJITC 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 genera- ting necessary register load/spill instructions. Besides, the native code generator also incorporates 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.

4.2 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.

4.2.1 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.

As a contrast, the bytecode sequence can be one-to-one translated into IR instructions as in Figure 1 (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 1 (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.

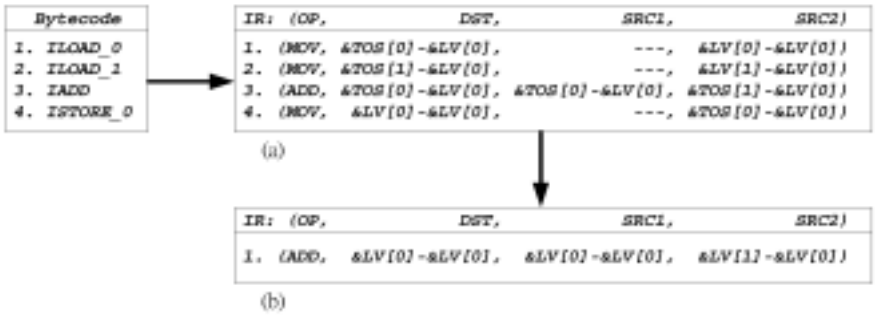

Figure 1 IR Generation (a) Without Optimization (b) With Instruction Folding

4.2.2 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 APPENDEX A

5. Results

This section presents the experiment results showing the effectiveness of the major optimizations employed in our Memory-Constrained JIT Compiler (MCJITC). We outline the experiment environment and benchmarks first, then present and discuss our measurement results.

5.1 Experiment Environment

Our MCJITC 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. 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 MCJITC, maximum optimization is specified with -O2 option, and other options remain default. Last but not least, our target architecture is ARM7TDMI.

5.2 Experiment Benchmarks

We choose Embedded CaffeineMark 3.0 [6] 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 1).

Table 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 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

The original design of Embedded CaffeineMark 3.0 is each test executes for a fixed amount of time, and the reported score is proportional 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.

5.3 Experiment Results

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.

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

Table 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	955,035,635	237,416,508	195,197,888	166,003,256
Logic	984,611,450	829,966,475	812,832,280	812,837,920
String	966,478,059	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 2 shows the speedup of the optimization setups over the pure interpreter, for ease of understanding. The key observation is that instruction folding for stack operations 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.

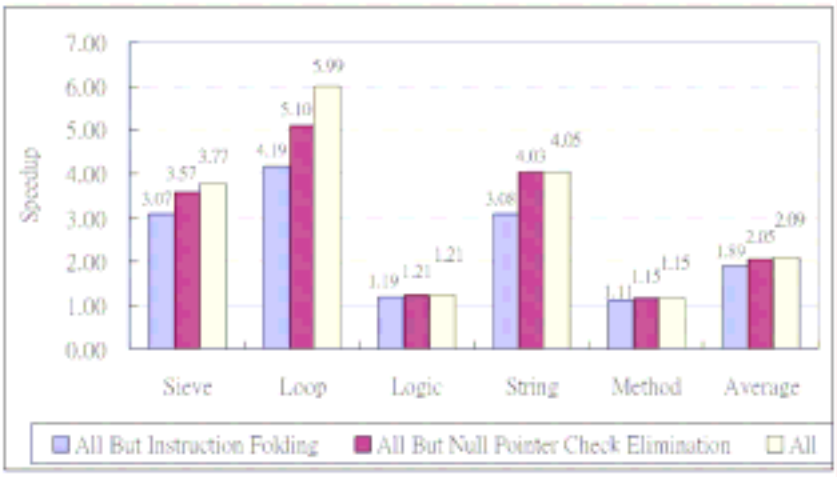

Figure 2. Effects of Optimizations

5.4 Conclusion

We have implement the Mix-Mode Memory-Constrain JIT Compiler (MCJITC) based on version 1.0.4 of Sun's KVM, and employ some optimizations including instruction folding of stack operations and rule-based null point check elimination in our MCJITC. According to the results shown in Figure 2, we see our MCJITC is average 2.09 faster over the pure interpreter, but we also observe that the performance improvement on the logic and method test is not significant. So, for future research directions, more optimizations may be incorporated into our MCJITC. For

example, one optimization that deserves most attention is method inlining, since there is not much optimization space for tiny method calls in our MCJITC, so employing method inlining can be effective under such circumstance.

三、參考文獻

[1] O. Agesen and D. Detlefs, "Mixed-mode Bytecode Execution," TR-2000-87, Sun Microsystems, June 2000
[2] 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
[3] 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
[4] 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
[5] "Connected Limited Device Configuration Specification," Verison 1.1, Sun Microsystems, March 2002
[6] Pendragon Software Corporation, *Embedded CaffeineMark 3.0 benchmark*, http://www.webfayre.com, 1997

四、計畫成果自評

We have completed the basic environment and some common optimizations of MCJITC this year. Experiment results show that our MCJITC is average 2.09 faster over the pure interpreter. For the research papers in the future, we will focus on some advanced designs and implementations of dual mode code generation, the use of multiple load/store operations and low-power issues on our MCJITC architecture.