OpenJIT Backend Compiler

Kouya SHIMURA

June 29, 1998

E-Mail: kouya@flab.fujitsu.co.jp
Computer Systems Lab.
FUJITSU LABORATORIES LTD.

1 Introduction

This document describes the overview and implementation of OpenJIT Backend Compiler.

This document covers the following:

- Goals of our project
- Implementation Overview
- Internal Specification
- Preliminary Results

2 Goals of our project

These are the goals for OpenJIT Backend System.

2.1 Target Environment

OpenJIT compiler is based on Sun's JDK version 1.1.X. Our target machine is a SPARC version 8 computer running Solaris 2.5 or later. Our code is almost written in java. Few code is written in C language for access to the structures inside of JDK and managing native code memory area.

2.2 Small Code Size

JVM has some complex instructions. In order to reduce compiled code size, these instructions are not inlined and translated to instruction sequence of runtime routine call.

2.3 Selectable Compiled Methods

A user can specify which methods should be compiled. It is specified by a environment variable before invoking Java. Classes and packages being compiled are also specified. Uncompiled methods are interpreted by JDK.

2.4 No Modifications to JDK

We doesn't modify Sun's JDK to integrate the JIT compiler. This will make porting to new versions of Sun's JDK faster and easier.

3 Implementation Overview

3.1 Integrating into JDK

Sun's JDK 1.1.X has the Java Native Code API. This is intended for a programmer to write native code generators inside the JVM. Using this API, we made success to integrate OpenJIT compiler into Sun's JDK without modification of JDK. OpenJIT-compiler is attached to JVM at runtime as one of dynamic link libraries.

3.2 Compilation Timing

In OpenJIT compiler, The unit of compilation is a method. When a class is loaded, each invoker of methods included in that class is hooked to compile it. When the method is invoked, OpenJIT compiler starts to work. After finishing to compile the method, compiler jumps to entry of the compiled method. If the compilation is failed for some reason – memory full, illegal bytecode, etc, such a method is interpreted by JDK.

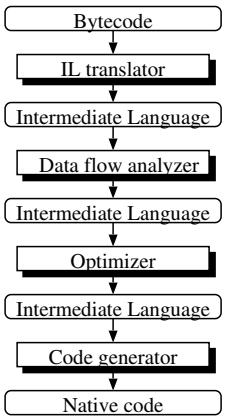
3.3 Compiled Code Area

Compiled code area is located at high address on memory and expanded upward. This area is allocated by mmap system call. Once a method is compiled, the compiled code are spooled on memory and reused. In current our implementation, this area never be freed. Managing compiled code area is one of our problems.

4 Internal Specification

4.1 Diagram of compiling flow

The following diagram shows the overall structure of the JIT compiler.



IL translator: Translate the bytecode into IL(Inermediate Language). One JVM instruction is expanded to several IL instructions.

Data flow analyzer: Analyze stack depth and verify the bytecode briefly. Moreover it determines operands data type in some instructions such as dup, dup2, etc.

Optimizer: Eliminate redundant instructions.

Code generator: Translate IL to native code.

4.2 Intermediate Language

The following is an instruction set of our intermediate language. Each instruction has 3 operands and each operands has data type tag. It has only 36 instructions and it will be portable to other hardware architectures.

```
nop     branch  call    return  tblsw   ret
move    add     addcc   addx    and     ld
ldsh    ldsb    lduh    ldd     or      div
sll     mul     sra     srl     st      stb
sth     sub     subcc   subx    xor     i2f
i2d     f2i     d2i     f2d     d2f     fneg
```

4.3 Optimization

OpenJIT compiler transform a sequence of bytecode to native code using simple optimization techniques. OpenJIT compiler uses the following two techniques.

- Register allocation

  Allocate operand stack and local variables to registers

- Peephole optimization

  Eliminate redundant instructions caused by register allocation

4.3.1 Example of register allocation

The following code is executing "`a=b+c`". The first column shows JVM's bytecode. The second column shows the naive code using stack. After allocating stack operands to registers, the code becomes like the third column. Some instructions loading from the stack or storing to the stack can be removed. Furthermore variables are allocated to registers, the code would be like the last column. The last code doesn't include any load and store instructions and can run faster.

bytecode	naive code	allocate stack	allocate vars
/ push var 1 on stack */*			
iload_1	→ ld [vars+0],%r1 st %r1,[optop+0]	→ ld [vars+0],%r1	→ mov %l1,%r1
/ push var 2 on stack */*			
iload_2	→ ld [vars+4],%r1 st %r1,[optop+4]	→ ld [vars+4],%r2	→ mov %l2,%r2
/ add top 2 entries and push the result */*			

iadd	→ ld [optop+0],%r1
	ld [optop+4],%r2
	add %r1,%r2,%r3
	st %r3,[optop+0]
		→ add %r1,%r2,%r1	→ add %r1,%r2,%r1

/ pop a value and write it into var 3 */*			
istore_3	→ ld [optop+0],%r1 st %r1,[vars+8]	→ st %r1,[vars+8]	→ mov %r1,%l3

4.3.2 Example of peephole optimization

The following code is executing "`a=b+100`". The first column shows JVM's bytecode. The second column shows the code after register allocation optimization. Instructions of SPARC can handle immediate value(signed 13 bit) and the code can be translated to the third one. Moreover register transferring instructions are redundant and the code is finally translated to only one instruction at the last column.

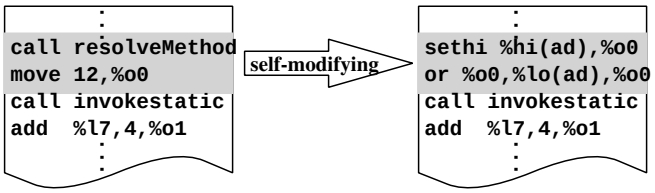
But these peephole optimization can be only applied within basic block.

bytecode	reg alloc	remove const	remove redundancy
/ push var 1 on stack */*			
iload_1	→ mov %l1,%r1	→ mov %l1,%r1	→ . . . . .
/ push constant 100 on stack */*			
ldc 100	→ mov 100,%r2	→	→ . . . . .
/ add top 2 entries and push the result */*			
iadd	→ add %r1,%r2,%r1	→ add %r1,100,%r1	→ add %l1,100,%l2
/ pop a value and write it into var 2 */*			
istore_2	→ mov %r1,%l2	→ mov %r1,%l2	→ . . . . .

4.3.3 Implementation of constant pool resolution

OpenJIT compiler compiles a method before execution and some part of compiled code possibly never be executed. In order to improve performance to execute constant pool resolution, OpenJIT compiler generates self-modifying code. At the compiling time, it generates calling sequence for *resolver* routine. When *resolver* is called, *resolver* routine overwrites caller code with code that executes setting address to a register.

The following figure shows an example of "`invokestatic #12`". In this case, the address `ad` is a pointer to method block and it is passed to `invokestatic` routine.



There is a problem that self-modifying code in a hardware can be costly, because it may require flushing the pipelines, instruction caches and so on. On the other hand, flushing will occur only once for each constant pool resolution used in a program. For constant pool resolution in loops, the flush cost is negligible if the loop iterates many times.

4.4 JIT Runtime routines

The following is the list of JVM's instructions using JIT runtime routines. *_quick* instructions are omitted but *_quick* variants are also included.

`anewarray`	athrow	checkcast	d2l
dcmpg	dcmpl	drem	f2l
fcmpg	fcmpl	frem	instanceof
invokeinterface	invokenonvirtual	invokestatic	invokevirtual
l2d	l2f	lcmp	ldiv
lmul	lrem	lshl	lshr
lushr	monitorEnter	monitorExit	multianewarray
newarray	newobject		

All the other stuff of JIT runtime routines are listed below.

function name	description
`arrayCheck`	check array boundary
`resolveClass`	for `anewarray`, `multianewarray`, `checkcast`, `instanceof`
`resolveField`	for `putfield`, `getfield`
`resolveMethod`	for invoking method
`resolveNew`	for `new`
`resolveStaticField`	for `putstatic`, `getstatic`
`resolveString`	for `ldc`

4.5 Exception handling

In spite of rarely exception happening, so much code for detecting exception have to be buried in compiled code that it make disadvantage of time and space. To avoid this, we use UNIX signal handling mechanism. The following two exceptions is caught by OS and handled.

- NullPointerException
- ArithmeticException(/ by zero)

For all other exceptions, it can be detected inside of compiled code or runtime routines and thrown to appropriate exception handler.

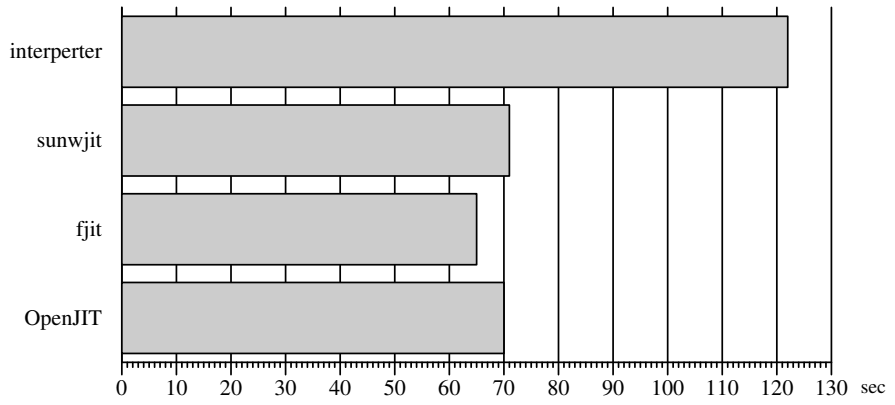
We made much effort to implement exception handling routine because it requires restoring context of processor. System call *setjmp* and *longjmp* on UNIX are easy to use for such purpose but too many *setjmp* call have to be buried in compiled code. Besides, when a synchronized method might be skipped by exception, monitor for synchronization have to be unlocked. Therefore we wrote very tricky exception handling routine in assembler.

5 Preliminary Results

For JDK's interpreter(no JIT) and Sun's JIT compiler (*sunwjit*), Fujitsu's JIT compiler written in C(*fjit*), and OpenJIT, we measured the performance. The machine we used for evaluation is Sun Ultra workstation(UltraSPARC-II 247MHz, memory: 1GB, OS:Solaris 2.6).

5.1 Speedups of javac

The following graph shows the CPU time to compile *javac* (Java compiler) itself.



5.2 Size of compiled code

We used *javac* to measure the size of compiled code. Additionally we also measure the number of instructions for bytecode and compiled code since the unit of instruction word is different. The result was further smaller than we expected.

	code size
bytecode	143KB (1.0)
native code	488KB(3.41)