Code Generation and Optimization for Java-to-C Compilers*

Youngsun Han, Shinyoung Kim, Hokwon Kim,
Seok Joong Hwang, and Seon Wook Kim

Department of Electronics and Computer Engineering
Korea University, Seoul, Korea

Abstract. Currently the Java programming language is popularly used in Internet-based systems, mobile and ubiquitous devices because of its portability and programability. However, inherently its performance is sometimes very limited due to interpretation overhead of class files by Java Virtual Machines (JVMs). In this paper, as one of the solutions to resolve the performance limitation, we present code generation and optimization techniques for a Java-to-C translator. Our compiler framework translates Java bytecode into C codes with preserving Java's programming semantics, such as inheritance, method overloading, virtual method invocation, garbage collection, and so on. Moreover, our compiler translates `for` in Java into `for` in C instead of `test` and `jump` for better performance. Our runtime library fully supports Connected Limited Device Configuration (CLDC) 1.0 API's.

1 Introduction

Java's platform independent architecture gives excellent portability. The applications written in Java can be compiled into location-independent codes moving on the Internet and running on every platform. In addition to the portability, its enhanced programability for more advanced development is one of the most important merits. However, despite of the distinguished advantages over other programming languages, there are two shortcomings to use Java, i.e, the size of Java virtual machines and performance limitation due to interpretation. The size of a full-featured JVM is too big to be used on small devices like mobile phones and PDAs. Due to the limited computing power resources on small embedded devices, a few different versions of JVMs have been proposed [1,2], and therefore, the class files cannot be executed on all kinds of client machines. Examples of the full-featured JVMs are JVM from Sun Microsystems [3], Jikes RVM [4], and an example of the partially-featured JVM due to the resource constraint is Java 2 Platform, Micro Edition (J2ME) [2].

The software interpretation incurs much higher runtime overhead than direct execution to use native codes. To alleviate the performance problem, many methods have been proposed such as just-in-time (JIT) and ahead-of-time (AOT)

* This research work (Ex : Paper, Patent, Standard, IP, SoC, IT System, etc) has been supported by Nano IP/SoC Promotion Group of Seoul R&BD Program in 2006.

X. Zhou et al. (Eds.): EUC Workshops 2006, LNCS 4097, pp. 785–794, 2006.

compilers. The just-in-time compiler [4,5,6] converts a sequence of bytecode (a method) into native codes at runtime, dynamically links, and executes them. This approach has been widely used for the last years. The alternative method is to translate class files into C codes on offline [7,8,9].

In this paper we present code generation and optimization techniques for a Java-to-C compiler. Also our compiler fully supports Connected Limited Device Configuration (CLDC) 1.0 API's. Moreover, our Java-to-C compiler translates `for` in Java into `for` in C in order to get better performance because gcc compiler cannot optimize `goto` statement generated for `for` in Java by our earlier Java-to-C compiler, but it can apply various optimization techniques to `for` in C.

Toba [8] is a system to generate standalone Java applications which were targeted for JDK 1.1. It has a bytecode-to-C translator and additional run-time libraries to support garbage collection, thread management and Java API. In [9], the Java-through-C compilation system for embedded systems has been developed. There is a definite difference between our Java-to-C compiler and the others. Our compiler only generates good quality of codes, to which a backend compiler (ex. `gcc` compiler) can effectively apply code optimization for improving performance.

The paper is organized as follows: we briefly present the structure of our Java-to-C compiler in Section 2, and the framework for C code emission from Java bytecode in Section 3. Also, In Section 4, we present the implementation of one optimization technique which generates `for` in C for `for` in Java instead of `test` and `jump`. Section 5 discusses issues about code generation and optimization with performance analysis, and finally Section 6 makes the conclusion.

2 Structure of Java-to-C Compiler

Our Java-to-C compiler is organized into three components: a Java decoder, a bytecode-to-C translator, and runtime libraries. The Java decoder analyzes the class files, and generates class blocks to maintain class information. The translator converts Java bytecode sequences into a sequence of C codes. Finally, the generated C codes are linked with runtime libraries to include routines for garbage collection, thread management, and Java CLDC 1.0 API in order to build executable codes. A thorough description of the structure is provided by our technical report [10].

3 Code Generation

There are a few issues in code generation, such as bytecode translation, exception handling, garbage collection, and thread management.

During translation, the preprocessor splits the whole bytecode sequence into several basic blocks to construct a control flow graph(CFG) for a method. The control flow graph is used to compute the stack state. The liveness of temporary variables are properly maintained through computing the stack state. Because bytecode translation is performed in compilation time, the information of the simulated operand stack is useful for code generation. After the preprocessor finds out basic blocks, bytecode-to-C translation is preformed for each basic block. The generated C codes for all basic blocks in a Java method are wrapped up in a switch statement like Toba [8].

The switch statement is used to handle Java exceptions. Some exceptions are specified to be thrown in Java virtual machine specification [11] when certain conditions are satisfied. These exceptions can be ignored according to the execution environment. The runtime program counter of the JVM is used to find a corresponding exception handler. A local variable `pc` is employed to mimic the program counter in the JVM. The variable is set at the beginning of every basic blocks. Additionally, C's `setjmp` and `longjmp` routines are used to handle exceptions.

An automatic garbage collection is supplied in Java. If certain objects are no longer referenced in a Java program, the objects will be de-allocated without any effort by a programmer. We use Boehm-Demers-Weiser conservative garbage collector [12].

Our system has been targeted to support the Connected Limited Device Configuration (CLDC) 1.0 [13]. Because the only java.lang.Thread class is specified in CLDC API, complex runtime libraries for java.lang.ThreadGroup are not implemented. The Java runtime package for thread management is implemented using Linux PThread libraries. A thread can hold only a lock associated with a monitor which all objects have competed. Synchronization between threads is guaranteed via monitors.

4 Optimization

4.1 Motivation

Figure 1 shows one of major Java methods in `LU` benchmark, and Figure 2 shows C output generated by our Java-to-C compiler and assembly output by `gcc` compiler. When the Java-to-C compiler translates Java `for` loop in forms of `test` and `jump` statements instead of `for` in C, `gcc` compiler sometimes does not assign a loop induction variable to a register. As shown in Figure 2, the loop induction variable *Lvi14* is incremented by `leal 1(%eax), %eax` instruction. It incurs huge processor stalls. When the Java-to-C compiler generates `for` in C for `for` in Java, we could get about 20% higher speedup than before, since `gcc` compiler applies loop optimizations to `for` in C. Therefore, we designed a state machine to identify `for` in Java bytecodes for generating `for` in C. Using gathered information, our compiler can translate `for` in Java in forms of `for` in C instead of `test` and `jump`.

4.2 Detection of `for` Loop in Java Bytecode

Our compiler detects Java `for` loops by using a state machine to recognize the following code sections in bytecode sequences: an initialization of an induction variable, a loop bound, a modification of an induction variable, and a backward

```
public static int factor(double ad[][], int ai[])
{
   ....
       for (int i2 = l + 1; i2 < i; i2++)
           ad2[i2] -= d3 * ad3[i2];
   ....
}
```

Fig. 1. Example of Java code in LU

```
_L229:                                        .L184:
    pc = 229;                                     cmpl    -204(%ebp), %eax
    i0 = Lvi14;                                   movl    $229, -188(%ebp)
    i1 = Lvi2;                                    jge     .L186
    if( i0 >= i1 ) goto _L257;                .L187:
_L235:                                            fld     %st(0)
    pc = 235;                                     fmull   16(%esi,%eax,8)
    a0 = Lva10;                                   fsubrl  16(%edx,%eax,8)
    i1 = Lvi14;                                   fstpl   16(%edx,%eax,8)
    a2 = a0;                                      leal    1(%eax), %eax
    i3 = i1;                                      movl    $235, -188(%ebp)
    d2 = ((struct darray*)a2)->data[i3];          jmp     .L184
    d3 = Lvd12;                               .L186:
    a4 = Lva11;
    i5 = Lvi14;
    d4 = ((struct darray*)a4)->data[i5];
    d3 = d3 * d4;
    d2 = d2 - d3;
    ((struct darray*)a0)->data[i1] = d2;
    Lvi14 += 1;
    goto _L229;
_L257:
```

Fig. 2. C and assembly outputs from the example Java code in Figure 1

`jump` instruction which is the end of `for` loop. The state diagram is shown in Figure 3. This state machine can recognize several `for` loop patterns in Java bytecodes, and the recognizable patterns are shown in Table 1.

Initialization of an induction variable: In the state machine of Figure 3, state `q0` is an initial state and state `accept` is a final state. In Java bytecode sequences, an induction variable can be either 1) a local variable or 2) a member field or a method of an object. In Case 1, bytecodes such as `iconst, bipush, sipush, load, invokestatic`, `getstatic` and so on, are used to initialize an induction variable. In Case 2, `aload` appears first, and then `arraylength, getfield, invokespecial, invokeinterface`, or `invokeivirtual` follows. If a compiler detects either case, the state machine goes to state `q1`. When the state machine is in state `q1` and the following instruction is `istore`, the state machine

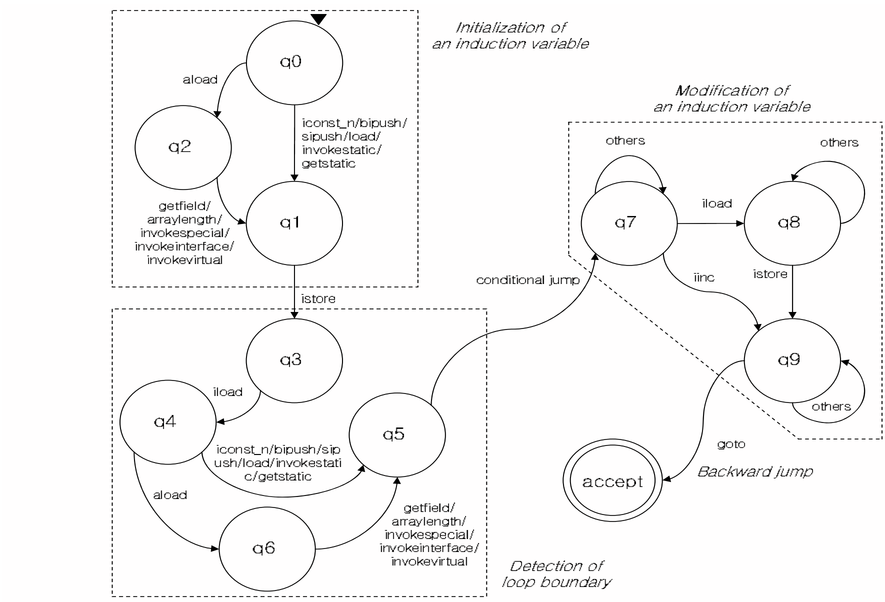

Fig. 3. State diagram to search `for` in Java bytecode

moves into state `q3`. This `istore` instruction stores the initialized induction variable into a memory. Thus, when the state machine reaches state `q3`, it decides that a code section of an induction variable initialization is found.

Loop bounds: In Java bytecode, after initialization of an induction variable, a code section of a comparison part should appear according to a syntax of `for` in Java. If each condition for the syntax is not satisfied, the state machine is moved into the initial state `q0`. To compare an induction variable with a loop bound, `iload` instruction is performed to read an induction variable from a memory. At this time, the state machine moves from state `q3` into state `q4`. After the induction variable is loaded from a memory, a loop bound variable is loaded. If the loop bound variable is a local variable, the state is moved to `q5` from `q4`, and if the loop bound variable is a member field or a method of an object, the state is moved to `q6` and `q5` from `q4`.

Modification of an induction variable: We define two patterns as modification of an induction variable in Table 1 for simple experiment. In the first pattern, we should find a pair of `iload` and `istore` instructions that access the same memory location before `goto` statement for a backward jump (`q7` → `q8` → `q9`). The kinds of integer arithmetic and logic operations between these two instructions are not important. All we need are the start and the end of a code section of modification of an induction variable. In the second pattern, the start and the end of a code section of modification of an induction variable are the same. When `iinc` instruction appears, the state machine moves from state `q7` into state `q9`. The only condition that `iinc` is an instruction for a code section

Table 1. `for` loop patterns which our compiler can recognize

for(A = B; A < C; A = A+D or A++) {···}			
Part of for loop	Code in C	Description	Structure of Java bytecode
Initialization of an induction variable	A = B	Value initializing an induction variable	iconst, fconst, lconst, dconst, bipush, sipush, iload, fload, lload, dload, getstatic, invokestatic, aload-arraylength, aload-getfield, aload-invokevirtual, aload-invokespecial, aload-invokeinterface
		store	istore, fstore, lstore, dstore
Comparison part	A < C	Load an induction variable	Iload, fload, lload, dload
		Variable compared with an induction variable	iconst, fconst, lconst, dconst, bipush, sipush, iload, fload, lload, dload, getstatic, invokestatic, aload-arraylength, aload-getfield, aload-invokevirtual, aload-invokespecial, aload-invokeinterface
		Conditional jump(compare)	If_icmpeq, if_icmpne, if_icmpge, if_icmpgt, if_icmple, if_icmplt
Body	{···}	Body of for loop	Various instructions can occur
Modification of an induction variable	A = A + D	Load an induction variable	iload, fload, lload, dload
		Modification	add or other various kinds of arithmetic and logic operations.
		Store the updated induction variable	istore, fstore, lstore, dstore
	A++	Autoincrement	iinc

of modification of an induction variable is that the `iinc` instruction should be followed by `goto` instruction for a backward jump.

backward jump: When the state machine is in state `q9`, it can be said that all needed information about `for` loop is found. Therefore, if the next instruction is `goto` that makes a backward jump to the start of a code section of a comparison part, the state machine goes into a final state. In order to justify a correct backward jump, it is checked whether a target address of the conditional `jump` is the next instruction of the backward `jump`.

Nested loops: In order to detect nested loops efficiently, we used a loop stack as a data structure. Whenever the state machine reaches to state `q7` (after detecting an induction variable and a loop bound), all collected information is pushed into the loop stack and the state machine returns to state `q0`. In this way, our state machine can find all the front parts of loop patterns we define while scanning the whole code once. When reaching to the end of a whole code, the stack has the information about front parts of all the loops we found. The state machine pops from the stack one by one and returns to state `q7`. The `pc` also goes back to the address of the conditional `jump` given by the popped information and the state machine starts checking whether those front parts of the loops have proper instructions for modification of an induction variables and backward `jump`.

4.3 Code Emission

Figure 4 (a) shows the Java source code and Java bytecode of an example of `for` loop patterns accepted by our state machine. And Figure 4 (b) shows our compiler's code generation without recognizing Java `for` loops.

(a) Java source code and bytecode

```
for(l1 = 8; l1<i; l1++)
    k+= l1;
======================
52:bipush        8
54:istore        10
56:iload         10
58:iload_3
59:icmpge        75
62:iload         4
64:iload         10
66:iadd
67:istore        4
69:iinc          10  1
72:goto          56
```

(b) Code generation using `test` and `jump`.

```
  i0 = 8;
  Lvi10 = i0;
_L56:
  i0 = Lvi10;
  i1 = Lvi3;
  if( i0 >= i1 ) goto _L75;
_L62:
  i0 = Lvi4;
  i1 = Lvi10;
  i0 = i0 + i1;
  Lvi4 = i0;
  Lvi10 += 1;
  goto _L56;
```

(c) `for` in C generated from `for`

```
  i0 = 8;
  Lvi10 = i0;
_L56:
  for( ;
    Lvi10 < Lvi3 ;
    Lvi10 += 1 ) {
_L62:
  i0 = Lvi4;
  i1 = Lvi10;
  i0 = i0 + i1;
  Lvi4 = i0;
  }
```

Fig. 4. An example of `for` loop

Our compiler is able to recognize a `for` loop in Java bytecodes and translates it into `for` in C. Figure 4 (c) presents the result obtained by applying this optimization technique to Java bytecode in Figure 4 (a).

For our experiment we implemented a simple case in Table 1. Especially, when a member field or a method of an object is used in code sections of initialization of an induction variable and a comparison part, translating `for` in Java into `for` in C is complicated due to an exception handling. Thus, we selected simple `for` loop patterns to be translated into `for` in C. Moreover, these patterns are the most frequently used patterns in codes. So, before generating `for` in C for all the `for` in Java, we checked the performance improvement of translating these patterns into `for` in C.

5 Performance Evaluation

5.1 Methodology

The performance of our Java-to-C compiler has been tested using Java SciMark 2.0 benchmarks [14] on Zeon 2.0 GHz processor and 256 MB of memory with Redhat Linux 8.0, and compared with gcj. The generated C codes were compiled by gcc 3.2 C compiler. The SciMark [14] is a composite Java benchmark measuring the performance of numerical codes occurring in scientific and engineering applications. The benchmark consists of the following five applications: Fast Fourier Transform, Gauss-Seidel relaxation, Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization. Table 2 summarizes the applications.

Table 2. Java SciMark 2.0 Benchmark

Application	Description
FFT	Fast Fourier Transform exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.
SOR	Jacobi Successive Over-relaxation exercises typical accesses patterns in finite difference applications, for example, solving Laplace's equation in 2D with Drichlet boundary conditions.
Monte Carlo	Monte Carlo integration exercises random-number generators, synchronized function calls, and function inlining.
SparseMM	Sparse matrix multiply exercises indirection addressing and non-regular memory references.
LU	dense LU matrix factorization exercises linear algebra kernels (BLAS) and dense matrix operations.

5.2 Performance Comparison

The relative speedup to gcj is shown in Figure 5. Our Java-to-C compiler shows lower performance than gcj for all applications except SOR. Especially, in the case of Monte Carlo Integration, the overhead for synchronization in a single-threaded application makes that our system has worse performance than gcj. In our compiler, even if an application is single-threaded, a monitor locking is enabled. Toba [8] can reduce the synchronization overhead, since the actual monitor locking is delayed until more than one thread are created. However the execution time is greatly improved by turning-off system-defined exception handling. The gcj compiler optimizes an exception handling by using aggressive optimizations. Moreover, by translating for in Java into for in C, we can get faster execution time and see the possibility of further improvement in speedup. We could achieve 6% in SparseMM and 4% in Monte Carlo more speedup in loop code generation than test and jump. Such a little improvement is due to the limitation of loop patterns to be translated into for in C in our experiment. Our Java-to-C compiler generates for in C when modification of an induction variable is performed only by iinc statement. However for loop recognized by our compiler and translated into for in C is not the critical loop for performance in FFT, SOR and LU applications.

In the case of FFT, we could get much better performance when all major for loops are changed into for in C. For the experiment we found out major loops of each benchmark program, and then translated all the Java for loops

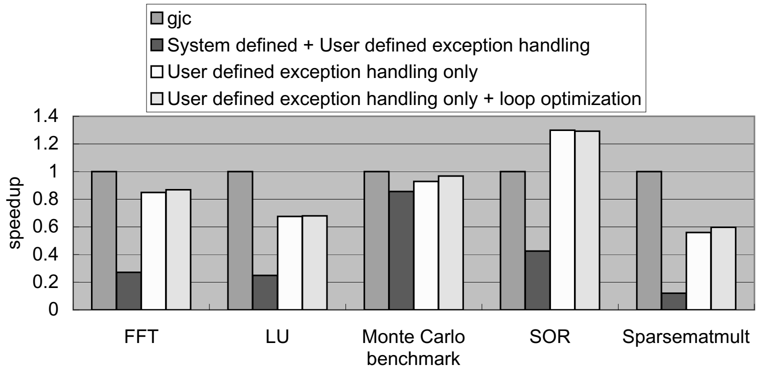

Fig. 5. Java SciMark 2.0 speedup

in `for` C by hand if our compiler did not translate them. The most critical method of FFT is `transform_internal` and it has four `for` loops. Our Java-to-C compiler translates one of them into `for` in C. When all the `for` loops are translated properly, the performance gets better up to 24% than test and jump code generation, and 5% faster than that `gcj`.

In the case of LU, the result is somewhat different. Although all the `for` loops in the most critical method of LU, `factor` were changed into `for` in C by hand, the performance was not improved. Therefore, we conclude that `for` loops in `factor` of LU are not an important factor in performance and loop optimization provided by gcc compiler is not very helpful in improving the performance. In the case of SOR, we found out that all the loops in the most critical method, `execute`, are translated in `for` in C already by our compiler. Therefore, we can conclude that in SOR, `for` loops are not the determining factor in overall performance.

6 Conclusion

In this paper, we presented the structure of the Java-to-C compiler to preserve the Java semantics, and discuss code generation and optimization issues. we presented the implementation of translating `for` in Java into `for` in C in order to make performance better. Because of limitation of loop patterns that can be translated `for` in C, we could not get as high performance as expected. However, by changing the rest `for` loops into `for` in C, we can discover the possibility of further improvement.

The generated C codes include many pointer and complex expressions, which prevent AOT compilers from applying advanced compiler optimization techniques like constant propagation, sub-expression elimination, inlining, and so on. In ongoing research, we have developed an IR framework between bytecode and C codes for helping AOT compilers generate better quality of codes and also have made our compiler generate `for` in C for more broad range of `for` loop patterns in Java source code.

References

1. Sun Microsystems: Java2 Platform, Standard Edition, (http://java.sun.com/j2se/)
2. Sun Microsystems: Java2 Platform, Micro Edition, (http://java.sun.com/j2me/)
3. Sun Microsystems: The Source for Java Technology. (http://java.sun.com/)
4. Bacon, D., Fink, S., Grove, D.: Space and time efficient implementation of the Java object model. In: European Conference on Object-Oriented Programming (ECOOP2002), Malaga, Spain (2002) 10–14
5. Undine, Kleine, A.: Tya, (http://sax.sax.de/˜ adlibit/)
6. Shudo, K.: shuJIT, (http://www.shudo.net/jit/#docs)
7. The Free Software Foundation: Guide to GNU gcj, (http://gcc.gnu.org/java/index.html)
8. Proebsting, T.A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T., Watterson, S.A.: Toba: Java for applications: A way ahead of time (WAT) compiler. In: Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS97). (1997)
9. Varma, A., Bhattacharyya, S.S.: Java-through-C compilation: An enabling technology for java in embedded systems. In: Design Automation and Test in Europe (DATE03), Paris, France (2004)
10. Han, Y., Kim, S., Kim, H., Hwang, S.J., Kim, S.W.: Code generation and optimization for java-to-c compiler. (http://compiler.korea.ac.kr/papers/java2c_report.pdf)
11. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1996)
12. Boehm, H.J., Demers, A., Weiser, M.: Boehm-Demers-Weiser conservative garbage collector. (http://www.hpl.hp.com/personal/Hans_Boehm/gc/)
13. Sun Microsystems: Connected Limited Device Configuration (CLDC), (http://java.sun.com/products/cldc/)
14. National Institute of Standards and Technology: SciMark 2.0. (http://math.nist.gov/scimark2/)