

Translating Java bytecode to X86 assembly code

Chun-Yuan Chen, Zong Qiang Chen, and Wu Yang
National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.

Abstract

ABSTRACT

Java has become one of the most popular languages for network applications. The main drawback of Java is concerned with the execution speed due to interpreted execution. We implemented a translation system that converts Java bytecode to X86 assembly code. The main approach of our translation system is that objects are still created and manipulated inside the (run-time) Java virtual machine. Other ordinary operations are translated into assembly code. Some pattern-based optimizations are applied during the translation process. Faster execution of Java programs will certainly make Java more attractive to network applications.

KEY WORDS

Java, assembly, translation, JIT, compilation, network application

1

1 Introduction

Java has become one of the most popular languages for network applications. The main drawback of Java is concerned with the execution speed due to interpreted execution. In order to speed up the execution of a Java program, we implemented a translation system that converts Java bytecode to X86 assembly code.

The main approach of the translation system is that objects are still created and manipulated inside the (run-time) Java virtual machine (JVM). This approach avoids the burden of garbage collection and most of the Java run-time security facilities are available. The overhead of this approach is that the resulting assembly program still makes frequent calls to JVM functions (through Java Native Interface).

¹The work reported here is supported by National Science Council, Taiwan, R.O.C., under grant NSC 93-2213-E-009-092. Due to the page limit of the conference, the sections on exceptions and ordinary instructions and a complete list of references are omitted. A 36-page full paper is available on the web at www.cis.nctu.edu.tw/wu-yang/papers/bytecode2X86.FULL.pdf.

An alternative is to create and manipulate objects purely in assembly code. This alternative approach may result in better performance. However, all the facilities provided by JVM cannot be utilized easily.

Our approach is an off-line translator, which allows the possibilities of thorough optimizations. In contrast, the just-in-time (JIT) compiler is a run-time translator, which can spend but a little time on optimization. Furthermore, the compilation result of JIT is not saved across execution. This means that repeated execution of the same bytecode incurs repeated JIT compilation, which is a serious overhead for frequently used programs. Many other research results are also focused on improving Java's performance.

Java virtual machine is a stack machine. We use the native stack in the X86 architecture to emulate the operand stack in JVM. Objects are allocated inside JVM heap. Auxiliary data structures created by our translator are allocated in the `.data` section. Constants, which are used in, for instance, the `ldc` instructions, in the constant pool of a `class` file are also allocated in the `.data` section. Registers in X86 architecture are used for scratch registers.

Our translator first scans the bytecode file to extract relevant information. The translator builds several tables for calculating addresses and for generating assembly instructions. In the simplest case, a bytecode instruction is translated into one or more assembly instructions. On the other hand, certain patterns—a sequence of 2 or more bytecode instructions—are recognized by our translator and they are translated into more efficient assembly instruction sequences.

The rest of the paper is organized as follows: Section 2 introduces operations for creating and manipulating objects. In section 3, we present a template-based optimization technique. Section 4 compares the performance of our system with that of Java interpreter and Just-In-Time (JIT) compiler.

2 Operations on Objects

2.1 Starting Up a JVM

In creating and manipulating objects, we need to use the services provided JVM. Therefore, we start up a JVM at the beginning of the resulting assembly program. The assembly program obtains a reference to the JVM when a JVM starts up. This reference is used to invoke JNI functions. Note that our translator allows mixed execution of assembly code and bytecode. JNI functions are the interface between the two forms of code.

2.2 Creating an Instance of a Class

The following Java code for creating an instance of a class is translated into the corresponding bytecode sequence:

```
StringBuffer x = new
StringBuffer(100);

new #2
dup
bipush 100
invokespecial #3
astore_1
```

where #2 and #3 are the entries of the constant pool `java/lang/StringBuffer` and `java/lang/StringBuffer<init>(I)V`, respectively.

In order to translate these bytecode instructions, we reserve an area in the `.data` area of which the layout is as follows:

```
index_2_clsname db
'java/lang/StringBuffer', 0
index_2_cls dword ?
index_3_midname db '<init>', 0
index_3_midsig db '(I)V', 0
index_3_mid dword ? --pointer to
the constructor
argreversebuf dword 20 dup(?)
```

Note that the number 3 in the names `index_3.mid`, etc. is determined by the the #3 argument in the bytecode instruction `invokespecial #3`. The same convention is followed through the generated X86 assembly code.

Then 3 JNI functions—`FindClass`, `GetMethodID`, `NewObject`—are called to create an instance. The created object resides in the heap of the JVM, not in the assembly code. The result is a pointer to the instance.

2.3 getfield and putfield

To access an instance variable `i` of object `x` (whose class is `fieldexample`) is done by the `getfield` and `putfield` instructions in bytecode.

```
fieldexample x=new
fieldexample();
x.i++;
```

The corresponding bytecode is as follows:

```
aload_1 ; x
dup
getfield #2 ; <Field int i>
iconst_1
iadd
putfield #2 ; <Field int i>
```

In order to translate the two bytecode instructions, an area in `.data` is reserved of which the layout is follows

```
index_2_clsname db
'fieldexample', 0
index_2_cls dword ?
index_2_fidname db 'i', 0
index_2_fidsig db 'I', 0
index_2_fid dword ?
argreversebuf dword 20 dup(?)
```

Since the created object resides in the heap of the JVM, we need to use JNI functions to access the fields of an object.

The instructions `getstatic` and `putstatic` are similar to `getfield` and `putfield`. The difference is that the call to JNI functions `GetFieldID`, `Get<Type>Field` and `Set<Type>Field` are replaced by `GetStaticFieldID`, `GetStatic<Type>Field` and `SetStatic<Type>Field`, respectively. Furthermore, we use the `jclass` reference, instead of the object reference, to access the static fields.

2.4 instanceof

The bytecode `instanceof` checks the class of an object.

```
if (a instanceof ABC) . . .

aload_1 ; a
instanceof #2 ; <Class ABC>
ifeq 15
```

The `instanceof` instruction is implemented with the JNI function `IsInstanceof`. The result (1 for yes and 0 for no) is pushed back on the stack.

2.5 Calling a Method

Java dynamically dispatches instance methods. Dynamic dispatching is implemented in X86 assembly code.²

There is one `allclassmethodtable` in the translator. Its layout is as follows:

```
classname idofclass superclass
methodname signature classname
nameInasm
methodname signature classname
nameInasm
. . .
Aclass 3 Bclass
xMethod xMethod-sig Aclass A_xMethod.1
yMethod yMethod-sig Bclass B_yMethod.1
zMethod zMethod-sig Bclass B_zMethod.2
Bclass 7 java.lang.Object
yMethod yMethod-sig Bclass B_yMethod.1
zMethod zMethod-sig Bclass B_zMethod.2
```

The `allclassmethodtable` lists all the classes in a program and all the methods defined in each class, including all the methods inherited from the superclasses. This table serves the purpose of dynamically dispatching an appropriate method.

In order to invoke a method, a structure, called the `objmapclass_table`, is reserved in the assembly code:

```
objmapclass_table struct
  jobj dword ?
  class dword ?
objmapclass_table ends

aa_objmapclass objmapclass_table
30 dup(<-1, -1>);
```

The `jobj` field is a reference to the object and the `class` field is the index of the entry of the object in the constant pool.

When a new object of an user defined class is created, the object reference and its index in the constant pool are saved in an `objmapclass_table` structure by the `build_objmapclasstable` procedure.

When an `invokevirtual` or `invokeinterface` instruction is translated into X86 assembly code, the translator will generate the following codes to find the `idofclass` of the created object.

²In bytecode, there are four instructions for invoking methods: `invokeStatic` is for static methods, `invokeSpecial` is for instance initialization method `<init>`, private instance methods, and methods of super class. `invokeInterface` and `invokeVirtual` are for instance methods, which are dispatched dynamically. The Java Virtual Machine uses a different instruction to invoke a method on an interface reference because it can not make as many assumptions about the method table offset given an interface reference as it can given a class reference[7].

```
push offset aa_objmapclass
push eax ; object reference is
saved in eax
call search_objmapclasstable
```

The `search_objmapclasstable` procedure searches the `aa_objmapclass` table and returns the `idofclass` of the created object. The translator then searches the `allclassmethodtable` with the method name and the signature. The result is a set of methods that match the name and signature of the called method and that depend on the `idofclass` as the entry point. According to this set, the translator generate a block of assembly code that selects the appropriate method from this set of methods.

For instance, suppose that class A is a subclass of class B. Both A and B define a method called `show`. The `show` method in A is renamed `A_show_2` while that in B is renamed `B_show_2` in the assembly code. The index of the entry of class A in the constant pool is 2 while that of B is 18. When a method `myObj.show(. . .)` is executed, the translator first determines the actual class of the object stored in `myObj` and then finds the index (stored in the `eax` register) of the entry of that class in the constant pool and generates the following X86 code:

```
.if eax == 2
call A_show_2
mov eax, 2
.endif
.if eax == 18
call B_show_2
mov eax, 18
```

In summary, for the `invokevirtual` and `invokeinterface` instructions, the translator uses the method name and method signature from constant pool table to search the `allclassmethodtable` for the appropriate methods, and generates assembly code for selectively calling that method.

On the other hand, for the `invokestatic` and `invokespecial` instructions, the methods that are actually invoked are statically bound (that is, decided at compile time). In particular, the called methods are not associated with object. Hence, the translator searches the `allclassmethodtable` directly for the appropriate method and generates assembly code to call that method directly.

2.6 Calling a Method in Bytecode Form

Calling a user-defined function (and dynamically dispatching the method) that has already been translated into X86 assembly is done completely in assembly code; it is not necessary to invoke JNI functions.

In order to run a Java program with our translation system, it is not necessary to translate the *whole* program before the program can be executed. We assume that part of the Java program, such as system APIs³, may stay in bytecode form. Thus, sometimes the translated assembly code may need to invoke methods in bytecode. This is done by invoking the JNI functions.

The translator first decides the class of the object. For the `invokestatic` or `invokespecial` instructions, the translator generates a call to the JNI function `FindClass` to get the class. For the `invokevirtual` or `invokeinterface` instructions, the translator generates a call to the JNI function `GetObjectClass` to get the class.

Then the translator decides the method to be called. For the `invokestatic` or `invokespecial` instructions, the translator generates a call to the JNI function `GetStaticMethodID` to get the method id. For the `invokevirtual` or `invokeinterface` instructions, the translator generates a call to the JNI function `GetMethodID` to get the method id.

Finally the translator generates a call to the proper method. For the `invokestatic` or `invokespecial` instructions, the translator generates a call to the JNI function `CallStatic<Type>Method`. For the `invokevirtual` or `invokeinterface` instructions, the translator generates a call to the JNI function `Call<Type>Method`. Here, the `<Type>` field, which is one of B, C, D, F, I, J, L, [, S, Z, and V, is determined from the type of the return value of the method.

2.7 The return Instructions

There are six kinds of return instructions in bytecode: `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, and `return`. The return instructions are translated into two X86 assembly instructions: `push` and `ret`.

3 Template-Based Optimization

In order to generate more efficient code, we also implemented a simple optimization scheme based on templates. Based on the concept in [4, 6], we developed about 20 templates for generating better code. For instance, the sequence of two bytecode instructions `bipush 9; istore_0` is translated into a single X86 assembly instruction `mov [aa_local_vars+0*4], 9`.

For example, the sequence of three bytecode instructions `iload_1; iload 8; if_icmpeq 38` is translated into the following four X86 instructions:

```
mov eax, [aa_local_vars+1*4]
mov ebx, [aa_local_vars+8*4]
cmp eax, ebx
je aa_38
```

For a second example, the following sequence of five bytecode instructions `iload 6; iload_3; iconst_1; iadd; if_icmpne 38` is translated into the following five X86 instructions:

```
mov eax, [aa_local_vars+6*4]
mov ebx, [aa_local_vars+3*4]
add ebx, 1
cmp eax, ebx
jne aa_38
```

When we consider two or more bytecode instructions together, we frequently can generate better Intel assembly code. We have incorporated two groups of templates in our translator system: templates that consists of 2 and 3 bytecode instructions, respectively.

4 Performance

We measured the performance of the assembly program on a platform comprised of AMD Athlon 1600 processor runs at 1.41 GHz with 256MB RAM, running Microsoft Windows XP professional edition. Java Virtual Machine is Sun J2SDK 1.4.1.01 for win32. The performance data is shown in Figure 1.

The overall performance of the resulting assembly program is between that of the pure Java interpreter (without JIT) and that of JIT.

5 Conclusion

We have implemented a system that translates bytecode into X86 assembly code.

To date, the speed of the assembly code generated by our translation system is faster than the interpreter, but is slower than that by JIT. Our translator could be improved further with more careful dynamic dispatching and better optimizations.

References

- [1] S.P. Dandamudi, Introduction to Assembly Language Programming, Springer, New York, 1998.
- [2] J. Dongarra, R. Wade, and P. McMahan, Linpack Benchmark – Java Version, available on the web at <http://www.netlib.org/benchmark/linpackjava/>, April 1996.

³In our current implementation, all the standard Java packages remain in bytecode form and all user packages are translated into X86 assembly code.

Benchmark	Interpreter	JIT	Asm
empty loop iterated 1,000,000,000 times	34.8	3.45	5.81
addition and subtraction on integer	13.47	0.84	2.05
multiplication and division on integer	16.5	3.7	5.08
addition and subtraction on long	14.41	0.72	5.52
multiplication and division on long (1)	22.09	9.88	105.99
multiplication and division on long (2)	22.09	9.88	35.25
addition and subtraction on float	11.67	1.81	3.69
multiplication and division on float	12.39	2.44	4.86
addition and subtraction on double	14.66	1.69	11.66
multiplication and division on double	15.58	3.19	12.73
array test 1	0.09	0.09	0.09
array test 2	12.28	12.06	12.48
array test 3	0.38	0.06	1.5
class field access	0.09	0.02	1.58
int field access	0.09	0.02	1.61
static method invocation	0.08	0.02	0.03
instance method invocation	0.08	0.02	0.08
Linpack	0.08	0.02	0.14
BTest	77	8	76

(1) Use software to emulate long multiplication and division.

(2) Convert long values to double, do the multiplication/division, then convert back to long.

Figure 1. Performance analysis

- [3] R. Grothmann, Java Benchmark, available on the web at <http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/java/bench/Bench.html>, 2003.
- [4] Ye Hua, Tong WeiQin, and Yao WenSheng, "Platform independence issue in compiling Java bytecode to native code," Proc. 4th International Conference on High Performance Computing in the Asia-Pacific Region, 530-532, May 2000.
- [5] K.R. Irvine, Assembly Language for Intel-Based Computers, 4th ed., Prentice-Hall, New Jersey, 2003.
- [6] H.D. Lambright, "Java Bytecode Optimizations," Proc. Compcon '97, 206-210, February 1997.
- [7] T. Linholm and Frank Yellin, The Java Virtual Machine Specification, 2nd ed., April 1999. Available on the internet at <http://java.sun.com/docs/books/cmspec/>.